Linguaggio C - Guida alla programmazione

Alessandro Bellini, Andrea Guidi



Prezzo Euro 24,50
ISBN: 88 386 0816-4
giugno 1999
420 pagine

Il linguaggio C, creato nel 1972 presso i Bell Laboratories, è attualmente il linguaggio di programmazione più utilizzato nei corsi universitari di introduzione all'informatica, grazie alla sua diffusione in ambito professionale, allo stretto legame con il mondo UNIX e alla facilità con cui consente il passaggio ad altri linguaggi quali C++, Java e Perl. In questo libro si illustrano dapprima in termini semplici e graduali, e con il continuo ausilio di esempi svolti, la sintassi e la semantica del linguaggio, per poi affrontare in profondità argomenti più avanzati, quali ricorsioni, archivi e strutture dati(pile, code, alberi e grafi);il capitolo conclusivo dedicato all'architettura Internet, introduce il programmatore C all'utilizzo del protocollo HTTP per la realizzazione di pagine Web dinamiche.

Indice

Autori

Alessandro Bellini è laureato in Ingegneria elettronica all'Università di Firenze ed è Dottore di ricerca in Ingegneria informatica e delle telecomunicazioni. E' stato professore incaricato di Ingegneria del Software all'Università di Siena e di linguaggi e traduttori all'Università di Firenze. Attualmente svolge attività professionale per enti pubblici, privati e nell'ambito di progetti di ricerca finanziati dall'Unione Europea.
E' autore di un testo sul linguaggio C edito da McGraw-Hill.
Andrea Guidi si è laureato in Scienze dell'informazione all'Università di Pisa ed è stato docente di Informatica presso l'Università dell'Ecuador. Ha lavorato come Project manager e come responsabile formazione nell'area database per importanti aziende del settore informatico. Attualmente è direttore commerciale di una società di commercializzazione software. E' autore di diversi volumi pubblicati da McGraw-Hill

Prefazione

Guida al linguaggio C, ha riscosso negli anni un vasto e duraturo successo in ambito Universitario; questa nuova edizione aggiorna il testo, approfondisce ulteriormente i temi trattati e lo arricchisce di nuovi argomenti, ormai divenuti urgenti, tra i quali l'uso dei protocolli di Internet.
L'obiettivo del testo resta quello di fornire una guida completa alla programmazione in linguaggio C, i concetti sono dapprima presentati con esemplificazioni e poi gradualmente sviluppati ed approfonditi. Per garantire un approccio morbido all'apprendimento del linguaggio, è stata posta grande attenzione all'ordine di presentazione degli argomenti. Sin dall'inizio il lettore può scrivere e provare sull'elaboratore programmi completi. L'esperienza insegna che uno dei modi migliori per apprendere un linguaggio è quello di introdurre pochi concetti alla volta e su quelli concentrarsi e svilupparli con esempi. La scomposizione dei programmi in funzioni segue i dettami di modularità, coesione ed accoppiamento richiesti dalla progettazione e programmazione strutturata; non si è voluto, comunque, inibire l'uso delle variabili globali usate con parsimonia e cautela.

I primi problemi da risolvere sono concettualmente semplici; il loro scopo è quello di prendere confidenza con i costrutti sintattici del linguaggio. I capitoli dall'uno al nove costituiscono le basi del C; inizialmente vi vengono introdotti: istruzioni, tipi dati, variabili e costanti, operatori, espressioni, strutture decisionali e iterative, istruzioni composte e annidate, librerie di funzioni. Successivamente vengono presentati gli array, le stringhe e vengono analizzati tutti i principali programmi di ricerca, ordinamento e fusione. Particolare cura viene infine impiegata nella trattazione della programmazione modulare: funzioni, passaggio di parametri, visibilità delle variabili. Ai puntatori, notoriamente ostici, è dedicato un intero ampio capitolo in cui vengono trattati con dovizia di esempi.

Nella seconda metà del testo si propongono allo studente temi e problemi più complessi che richiedono un'analisi preliminare, per valutare soluzioni alternative. Il capitolo dieci, dedicato alla ricorsione, presemta numerosi interessanti esempi di calcolo combinatorio (fattoriale, disposizioni, combinazioni), mentre i due capitoli successivi riprendono e approfondiscono tipi dati, costanti, operatori ed espressioni. In particolare vi vengono trattate le operazioni bit a bit tanto importanti per la programmazione di basso livello per la gestione di dispositivi hardware per cui il C è spesso utilizzato. Nel capitolo tredici, che si occupa di archivi, viene data una nuova soluzione al problema della "gestione di un'anagrafica", già precedentemente affrontato nel capitolo sui tipi dati derivati, questa volta utilizzando i file. Nei capitoli quattordici e quindici vengono trattate le strutture dati come pile, code, alberi e grafi, presentado differenti implementazioni che coinvolgono array, liste lineari, liste multiple e soluzioni miste. Vengono presi in esame problemi come la creazione, l'inserzione, l'eliminazione di elementi. In tal modo si raggiunge un duplice scopo. Da una parte offrire un valido banco di prova per il programmatore C, spingendolo a sfruttare caratteristiche tipiche del linguaggio come puntatori, strutture, funzioni di allocazione di aree memoria e complesse chiamate di funzioni. Dall'altro costruire una introduzione completa alle strutture dati che spesso vengono studiate solo sul piano teorico nei corsi di informatica. L'ultimo capitolo, dedicato all'architettura Internet, introduce il programmatore C all'utilizzo del protocollo HTTP per la realizzazione di pagine Web statiche e dinamiche.

Alcuni temi ed esemplificazioni percorrono l'intero testo, per esempio del problema della "gestione di una sequenza" viene data una prima soluzione nel capitolo sette con l'uso di funzioni, una seconda nel capitolo nove arricchita dalla disponibilità dei puntatori, infine una terza nel capitolo quattordici utilizzando una lista lineare. Ogni capitolo è concluso dalla presentazione di numerosi esercizi per la verifica del corretto apprendimento degli argomenti. Le soluzioni degli esercizi sono accompagnate da rifessioni e dal confronto di più alternative. La versione del linguaggio a cui si è fatto riferimento è quella dello standard internazionale ANSI.

Auguriamo di cuore buono studio/lavoro al lettore, che se lo desidera, può inviarci commenti e consigli ai seguenti indirizzi di posta elettronica `andreag@softpi.it` e `abel@mathema.com`.

1. Linguaggio C

Nel 1972 Dennis Ritchie progettava e realizzava la prima versione del linguaggio C presso i Bell Laboratories. Ritchie aveva ripreso e sviluppato molti dei costrutti sintattici del linguaggio BCPL, di Martin Richards, e del linguaggio B, di Ken Thompson, l'autore del sistema operativo UNIX. Successivamente gli stessi Ritchie e Thompson riscrissero in C il codice di UNIX.

Il C si distingueva dai suoi predecessori per il fatto di implementare una vasta gamma di tipi di dati - carattere, interi, numeri in virgola mobile, strutture - non originariamente previsti dagli altri due linguaggi. Da allora ad oggi non ha subito profonde trasformazioni: la sua sintassi è stata estesa, soprattutto in conseguenza della programmazione orientata agli oggetti (C++), ma nella sostanza e nello spirito il linguaggio è rimasto quello delle origini. Il C è un linguaggio di alto livello che possiede un insieme ristretto di costrutti di controllo e di parole chiave, ed un ricco insieme di operatori. Consente di programmare in modo modulare, per mezzo delle funzioni e delle macro, anche se non esiste una gerarchia di funzioni come, ad esempio, in Pascal. Pur essendo un linguaggio ad alto livello permette operazioni di basso livello tipiche del linguaggio macchina: si può, ad esempio, indirizzare la memoria in modo assoluto, funzionalità fondamentale per lo sviluppo di applicazioni di basso livello. E' un linguaggio apparentemente povero: non possiede istruzioni di entrata/uscita, ne istruzioni per operazioni matematiche. Ogni funzione diversa dai costrutti di controllo o dalle operazioni elementari sui tipi dati è affidata ad un insieme di librerie esterne. In questo modo, Ritchie riuscì a raggiungere due obiettivi: da una parte, mantenere compatto il linguaggio, dall'altra, poter estenderne le funzionalità semplicemente aggiungendo nuove librerie o ampliando quelle esistenti. E' stato talvolta definito come "il linguaggio di più basso livello tra i linguaggi di alto livello". Infatti, come abbiamo detto, nasce per lo sviluppo di sistemi operativi, quindi per software di basso livello, ma preservando la semplicità d'uso dei linguaggi della terza generazione.

Sono molti i fattori che hanno determinato la sua capillare diffusione. Il trampolino di lancio è stato il sistema operativo Unix. Il C ne ha seguito le sorti fin dall'inizio divenendo ben presto il linguaggio di programmazione preferito dalle università e dagli istituti di ricerca. Unix è stata la dimostrazione pratica della bontà e forza del linguaggio C. Il mondo dell'industria informatica lo ha notato ed oggi praticamente non esiste prodotto commerciale di larga diffusione - database, wordprocessor, foglio elettronico, browser etc. - che non sia scritto in C. Un'altro fattore che ha contribuito al successo del C è stato il personal computer. Quelle che erano funzioni di programmazione di sistema fino a qualche anno fa riservate a pochi specialisti oggi sono accessibili a tutti. Ad esempio, oggi è molto facile, anche per un programmatore C dilettante, pilotare direttamente l'hardware. In pratica il C sta sostituendo l'assembler. Esistono, infatti, dei compilatori talmente evoluti, che il codice assembler equivalente al C prodotto dal compilatore è talmente efficiente e compatto da risultare migliore di quello scritto anche da un buon programmatore assembler. I nuovi linguaggi che si sono presentati sulla scena dell'informatica, quali Java e Perl, devono molto al C, la cui conoscenza costituisce un ottimo punto di partenza per il loro apprendimento. L'enorme diffusione raggiunta e la sua efficienza, anche nel pilotare direttamente l'hardware, continuano a fare del C una scelta largamente condivisa anche per realizzare applicazioni Internet o parti di esse. Infine esistono anche motivi estetici. Per l'eleganza della sintassi e la compattezza dei costrutti, il C è una sfida permanente alle capacità intellettuali del progettista software, è anche una utilissima palestra per il giovane programmatore che con esso impara a risolvere una vasta classe di problemi, lo spinge a migliorare le proprie tecniche, e lo abitua a controllare le idiosincrasie della macchina senza dover ricorrere all'assembler.

1.1 Programmi

Iniziamo lo studio del C osservando il Listato 1.1, in cui possono già essere evidenziate alcune delle caratteristiche comuni alla struttura di ogni programma. La sua esecuzione inizia da `main()`; il *corpo del programma*, racchiuso tra parentesi graffe, è composto da una serie di istruzioni `printf` che verranno eseguite sequenzialmente. Ogni istruzione deve terminare con un carattere di punto e virgola.

```
#include <stdio.h>

main()
{
  printf("Tre");
  printf(" casettine");
  printf(" dai");
  printf(" tetti");
  printf(" aguzzi");
}
```

Listato 1.1 Un programma in linguaggio C

L'istruzione `printf` permette la stampa su video di ciò che è racchiuso tra parentesi tonde e doppi apici. Per esempio

```
printf("Tre");
```

visualizza:

```
Tre
```

Per poter utilizzare `printf`, così come le altre funzioni di input/output, si deve inserire all'inizio del testo la linea

```
#include <stdio.h>
```

che avverte il compilatore di includere i riferimenti alla libreria standard di input/output (`stdio` sta appunto per *standard input/output*). Le istruzioni vengono eseguite una dopo l'altra nell'ordine in cui si presentano. Il programma del Listato 1.1 è composto da tutte istruzioni `printf` e la sua esecuzione visualizzerà la frase

```
Tre casettine dai tetti aguzzi
```

Le istruzioni `printf` successive alla prima iniziano a scrivere a partire dalla posizione del video che segue quella occupata dall'ultimo carattere visualizzato dalla `printf` immediatamente precedente. Abbiamo perciò inserito all'interno degli apici uno spazio bianco iniziale; se non lo avessimo fatto avremmo ottenuto:

```
Trecasettinedaitettiaguzzi
```

Infatti anche lo spazio è un carattere come gli altri e, se vogliamo visualizzarlo, dobbiamo esplicitamente inserirlo nella giusta posizione. Se si desidera che l'uscita di ogni istruzione `printf` venga prodotta su una linea separata, si deve inserire `\n` al termine di ogni stringa e prima della chiusura dei doppi apici (vedi Listato 1.2). L'esecuzione del programma provocherà la seguente visualizzazione:

```
Tre
      casettine
      dai
      tetti
      aguzzi
```

In effetti, la sequenza `\n` corrisponde a un solo carattere, quello di linea nuova (*newline*). Dalla seconda riga in poi il primo carattere visualizzato è uno spazio: se vogliamo toglierlo dobbiamo cancellarlo dalla parte compresa tra apici nelle `printf` corrispondenti.

```
#include <stdio.h>

main()
{
  printf("Tre\n");
  printf(" casettine\n");
  printf(" dai\n");
  printf(" tetti\n");
  printf(" aguzzi\n");
}
```

Listato 1.2 Una variante del programma precedente

Il C distingue tra lettere maiuscole e minuscole; dunque si deve fare attenzione: se per esempio si scrive `MAIN()` o `Main()`, non si sta facendo riferimento a `main()`.

1.2 Variabili e assegnamenti

Supponiamo di voler calcolare l'area di un rettangolo di base 3 e altezza 7; osserviamo nel Listato 1.3 il programma che risolve il problema.

```
/* Calcolo area rettangolo */

#include <stdio.h>

main()
{
  int base;
  int altezza;
  int area;

  base = 3;
  altezza = 7;
  area = base*altezza;

  printf("%d\n", area);
}
```

Listato 1.3 Uso di variabili

Per rendere evidente la funzione espletata dal programma abbiamo inserito un commento:

```
/* Calcolo area rettangolo */
```

I commenti possono estendersi su più linee e apparire in qualsiasi parte del programma; devono essere preceduti da `/*` e seguiti da `*/`: tutto ciò che appare nelle zone così racchiuse non viene preso in considerazione dal compilatore e non ha nessuna influenza sul funzionamento del programma. Un altro modo per inserire un commento è farlo precedere da `//`, ma in questo caso deve terminare a fine linea:

```
// Calcolo area rettangolo
```

Dopo il `main()` e la parentesi graffa aperta sono presenti le *dichiarazioni* delle variabili (intere) necessarie:

```
int base;
int altezza;
int area;
```

La parola chiave `int` specifica che l'identificatore che lo segue si riferisce a una variabile numerica di tipo intero; dunque `base`, `altezza` e `area` sono variabili di questo tipo. Anche le dichiarazioni – così come le altre istruzioni – devono terminare con un punto e virgola. Nel nostro esempio, alla dichiarazione della variabile corrisponde anche la sua *definizione*, la quale fa sì che le venga riservato uno spazio in memoria centrale. Il *nome* di una variabile la identifica, il suo *tipo* ne definisce la dimensione e l'insieme delle operazioni che si possono effettuare su di essa. La dimensione può variare rispetto all'implementazione; alcune versioni del C riservano agli `int` uno spazio di quattro byte, il che permette di poter lavorare su interi che vanno da –2147483648 a +2147483647; altre versioni riservano due byte (gli interi permessi vanno da –32768 a +32767) . Tra le operazioni fra `int` consentite vi sono: somma, sottrazione, prodotto e divisione, che corrispondono rispettivamente agli operatori `+`,`-`,`*`,`/`.

L'istruzione:

```
base = 3;
```

assegna alla variabile `base` il valore 3; inserisce cioè il valore (3) che segue l'operatore = nello spazio di memoria riservato alla variabile (`base`). Effetto analogo avrà `altezza = 7`. L'assegnamento è dunque realizzato mediante l'operatore =.

L'istruzione

```
area = base*altezza;
```

assegna alla variabile `area` il prodotto dei valori di `base` e `altezza`, mentre l'ultima istruzione,

```
printf("%d\n", area);
```

visualizza 21, il valore della variabile `area`. Tra i doppi apici, il simbolo di percentuale `%` specifica che il carattere che lo segue definisce il formato di stampa della variabile `area`; `d` (*decimal*) indica che si desidera la visualizzazione di un intero nel sistema decimale. Invece `\n` provoca come abbiamo già visto un salto a linea nuova dopo la visualizzazione.

In generale la struttura di un programma C prevede che le variabili possano essere dichiarate sia dopo `main()` e la parentesi graffa aperta, e anteriormente alle istruzioni operative come nell'esempio visto, sia prima di `main()`. La struttura generale risulta quindi la seguente:

```
inclusione librerie
dichiarazioni di variabili
main
{
  dichiarazioni di variabili
  istruzione 1
  istruzione 2
  istruzione 3
  ...
  istruzione N
}
```

Si tenga presente che nella sintassi il punto e virgola fa parte dell'istruzione stessa.

Le dichiarazioni delle variabili dello stesso tipo possono essere scritte in sequenza separate da una virgola; per esempio, nel Listato 1.3 avremmo potuto scrivere:

```
int   base, altezza, area;
```

Dopo la dichiarazione di tipo sono specificati gli *identificatori* di variabile, che possono essere in numero qualsiasi, separati da virgola e chiusi da un punto e virgola. In generale, quindi, la dichiarazione di variabili ha la forma:

```
tipo  lista di identificatori;
```

Esistono inoltre regole da rispettare nella costruzione degli identificatori, che devono iniziare con una lettera o con un carattere di sottolineatura _ e possono contenere lettere, cifre e _. La lunghezza può essere qualsiasi ma caratteri significativi sono spesso i primi 255 (247 secondo lo standard), anche se nelle versioni del C meno recenti questo limite scende a 32 o anche a 8 caratteri. Le lettere maiuscole sono considerate diverse dalle corrispondenti minuscole. Esempi di identificatori validi sono: `nome1`, `cognome2`, `cognome_nome`, `alberoBinario`, `volume`, `VOLUME`, `a`, `b`, `c`, `x`, `y`; al contrario non sono corretti: `12nome`, `cognome-nome`, `vero?` e `padre&figli`. Teniamo a ribadire che `volume` e `VOLUME` sono differenti. Oltre a rispettare le regole precedentemente enunciate, un identificatore non può essere una parola chiave del linguaggio (vedi Appendice B per l'elenco delle parole chiave), né può essere uguale a un nome di funzione.

Allo scopo di rendere più chiaro il risultato dell'esempio precedente, si possono visualizzare i valori delle variabili `base` e `altezza`:

```
printf("%d ", base);
printf("%d ", altezza);
printf("%d", area);
```

Nelle prime due istruzioni `printf` si è inserito all'interno dei doppi apici, di seguito all'indicazione del formato di stampa `%d`, uno spazio, in modo che venga riportato in fase di visualizzazione dopo il valore della base e dell'altezza, così da ottenere:

```
3 7 21
```

e non `3721`. Se si vuole far precedere la visualizzazione dei valori da un testo di descrizione, è sufficiente inserirlo prima del simbolo di percentuale:

```
printf("Base: %d ", base);
printf("Altezza: %d ", altezza);
```

```
printf("Area: %d", area);
```

Quello che viene prodotto in esecuzione è

```
Base: 3 Altezza: 7 Area: 21
```

Per fare in modo che a ogni visualizzazione corrisponda un salto riga si deve inserire `\n` prima della chiusura dei doppi apici:

```
printf("Base: %d\n", base);
printf("Altezza: %d\n", altezza);
printf("Area: %d\n", area);
```

In questo caso in esecuzione si otterrebbe

```
Base: 3
Altezza: 7
Area: 21
```

Mentre `int` è una parola chiave del C e fa parte integrante del linguaggio, `base`, `altezza` e `area` sono identificatori di variabili scelti a nostra discrezione. Lo stesso effetto avremmo ottenuto utilizzando al loro posto altri nomi generici, quali `x`, `y` e `z`.

La forma grafica data al programma è del tutto opzionale; una volta rispettata la sequenzialità e la sintassi, la scrittura del codice è libera. In particolare, più istruzioni possono essere scritte sulla stessa linea, come nell'esempio seguente:

```
#include <stdio.h>
main() {int x,y,z; x = 3; y = 7;
z= x*y; printf("Base: %d\n", x); printf("Altezza: %d\n", y);
printf("Area: %d\n", z);}
```

Questo programma, però, è notevolmente meno leggibile del precedente.

✓ **NOTA**

Lo stile facilita il riconoscimento delle varie unità di programma e riduce il tempo per modificare, ampliare e correggere gli errori. Se ciò è vero in generale, lo è particolarmente per questo linguaggio poiché, come si avrà modo di vedere, il C spinge il programmatore alla sintesi, all'utilizzo di costrutti estremamente asciutti, essenziali. Non importa quale stile si decida di utilizzare, importante è seguirlo con coerenza.

In generale è bene dare alle variabili nomi significativi, in modo che si possa facilmente ricostruire l'uso che si è fatto di una certa variabile, qualora si debba intervenire a distanza di tempo sullo stesso programma.

1.3 Costanti

Nel programma per il calcolo dell'area visto nel paragrafo precedente, i valori di `base` e `altezza` sono costanti, poiché non variano durante l'esecuzione del programma stesso. Evidentemente avremmo potuto scrivere direttamente

```
area = 3*7;
```

Quando un certo valore viene utilizzato in modo ricorrente è opportuno rimpiazzarlo con un nome simbolico; per farlo dobbiamo definire all'inizio del programma, mediante l'istruzione `define`, un identificatore di costante in corrispondenza del valore desiderato:

```
#define BASE 3
```

Grazie a questa istruzione, all'interno del programma potremo utilizzare `BASE` al posto del valore intero 3. La definizione stessa di costante implica che il suo valore non può essere modificato: `BASE` può essere utilizzata in un'espressione a patto che su di essa non venga mai effettuato un assegnamento. Il programma del paragrafo precedente potrebbe quindi essere trasformato in quello del Listato 1.4.

```
/* Calcolo area rettangolo, prova utilizzo costanti */

#include <stdio.h>

#define BASE 3
#define ALTEZZA 7

main()
{
  int area;

  area = BASE * ALTEZZA;
  printf("Base: %d\n", BASE);
  printf("Altezza: %d\n", ALTEZZA);
  printf("Area: %d\n", area);
}
```

Listato 1.4 Uso di costanti

Un nome di costante può essere un qualsiasi identificatore valido in C. Abbiamo scelto di utilizzare esclusivamente caratteri maiuscoli per le costanti e caratteri minuscoli per le variabili per distinguere chiaramente le une dalle altre. Le costanti `BASE` e `ALTEZZA` vengono considerate di tipo intero in quanto il loro valore è costituito da numeri senza componente frazionaria.

Invece di utilizzare direttamente i valori, è consigliabile fare uso degli identificatori di costante, che sono descrittivi e quindi migliorano la leggibilità dei programmi. Per fare in modo che il programma precedente calcoli l'area del rettangolo con base 102 e altezza 34, è sufficiente modificare le linee dov'è presente l'istruzione `define`:

```
#define BASE 102
#define ALTEZZA 34
```

L'uso delle costanti migliora due parametri classici di valutazione dei programmi: flessibilità e manutenibilità. La `define` è una macroistruzione (*macro*) del precompilatore C che, come si vedrà, offre altre possibilità oltre a quella di poter definire costanti.

1.4 Input e output

Perché il programma per il calcolo dell'area del rettangolo sia più generale ed effettivamente "utile", l'utente deve poter immettere i valori della base e dell'altezza, mediante l'istruzione di input:

```
scanf("%d", &base);
```

L'esecuzione di questa istruzione fa sì che il sistema attenda in input un dato da parte dell'utente. Analogamente a quello che accadeva in `printf`, `%d` indica un valore intero in formato decimale che verrà assegnato alla variabile `base`. Si presti attenzione al fatto che in una istruzione `scanf` il simbolo `&` (*ampersand*) deve precedere immediatamente il nome della variabile; `&base` sta a indicare l'indirizzo di memoria in cui si trova la variabile `base`. L'istruzione `scanf("%d", &base);` può allora essere così interpretata: "leggi un dato intero e collocalo nella posizione di memoria il cui indirizzo è `&base`".

Durante l'esecuzione di un programma può essere richiesta all'utente l'immissione di più informazioni, perciò è opportuno visualizzare delle frasi esplicative; a tale scopo facciamo precedere le istruzioni `scanf` da appropriate visualizzazioni in output sul video, tramite istruzioni `printf`:

```
printf("Valore della base: ");
scanf("%d", &base);
```

L'argomento di `printf` è semplicemente una costante, quindi deve essere racchiuso tra doppi apici. Quello che apparirà all'utente in fase di esecuzione del programma sarà:

```
Valore della base:
```

In questo istante l'istruzione `scanf` attende l'immissione di un valore; se l'utente digita 15 seguito da Invio:

```
Valore della base: 15<Invio>
```

questo dato verrà assegnato alla variabile `base`. Analogamente, possiamo modificare il programma per l'immissione dell'altezza e magari aggiungere un'intestazione che spieghi all'utente cosa fa il programma, come nel Listato 1.5.

```
/* Calcolo area rettangolo */

#include <stdio.h>

int base, altezza, area;

main()
{
  printf("AREA RETTANGOLO\n\n");

  printf("Valore base: ");
  scanf("%d", &base);
  printf("Valore altezza: ");
  scanf("%d", &altezza);

  area = base*altezza;

  printf("Base: %d\n", base);
  printf("Altezza: %d\n", altezza);
  printf("Area: %d\n", area);
}
```

Listato 1.5 Immissione di valori

Vediamo il risultato dell'esecuzione del programma nell'ipotesi che l'utente inserisca i valori 10 e 13:

```
AREA RETTANGOLO

Valore base: 10
Valore altezza: 13
Base: 10
Altezza: 13
Area: 130
```

Per lasciare una linea vuota si deve inserire un ulteriore `\n` nell'istruzione `printf` all'interno di doppi apici: `printf("AREA RETTANGOLO\n\n")`. Il primo `\n` fa andare il cursore a linea nuova dopo la visualizzazione di

AREA RETTANGOLO, il secondo lo fa scorrere di un ulteriore linea. Il ragionamento è valido in generale: se si desidera saltare un'altra riga basta aggiungere un \n e se si vuole lasciare una linea prima della visualizzazione si fa precedere \n ad AREA RETTANGOLO:

```
printf("\nAREA RETTANGOLO\n\n\n");
```

Inoltre è possibile inserire il salto in qualsiasi posizione all'interno dei doppi apici, come nel seguente esempio:

```
printf("AREA \nRET\nTAN\nGOLO");
```

che provoca in fase di esecuzione la visualizzazione:

```
AREA
RET
TAN
GOLO
```

Si possono stampare più variabili con una sola printf, indicando prima tra doppi apici i formati in cui si desiderano le visualizzazioni e successivamente i nomi delle rispettive variabili. L'istruzione

```
printf("%d %d %d", base, altezza, area);
```

inserita alla fine del programma precedente stamperebbe, se i dati immessi dall'utente fossero ancora 10 e 13:

```
10 13 130
```

Nell'istruzione il primo %d specifica il formato della variabile base, il secondo %d quello di altezza e il terzo quello di area. Per raffinare ulteriormente l'uscita di printf, si possono naturalmente inserire degli a-capo a piacere:

```
printf("%d\n%d\n%d", base, altezza, area);
```

che hanno come effetto

```
10
13
130
```

Se, per esempio, si desidera una linea vuota tra il valore della variabile base e quello di altezza e due linee vuote prima del valore della variabile area, è sufficiente inserire i \n nella descrizione dei formati, esattamente dove si vuole che avvenga il salto a riga nuova:

```
printf("%d\n\n%d\n\n\n%d", base, altezza, area);
```

All'interno dei doppi apici si possono scrivere i commenti che devono essere stampati. Per esempio, se la visualizzazione della terza variabile deve essere preceduta da Area:, l'istruzione diventa la seguente:

```
printf("%d\n%d\nArea: %d", base, altezza, area);
```

che darà in uscita

```
10
13
Area: 130
```

Analogamente si può procedere con le altre variabili:

```
printf("Base: %d\nAltezza: %d\nArea: %d", base, altezza, area);
```

Si tratta dunque di inserire i vari messaggi che devono apparire sul video tra doppi apici, prima o dopo i simboli che descrivono i formati degli oggetti da visualizzare.

Così come \n effettua un salto a linea nuova, la sequenza \t provoca l'avanzamento del cursore di uno spazio di tabulazione:

```
printf("Base: %d\tAltezza: %d\tArea: %d", base, altezza, area);
```

produce come uscita

```
Base: 10          Altezza: 13       Area: 130
```

Esistono altre sequenze di caratteri con funzioni speciali, dette *sequenze di escape*. Riassumiamo quelle più usate, invitando il lettore a provarle nelle `printf`.

`\n` va a linea nuova
`\t` salta di una tabulazione
`\b` ritorna un carattere indietro (*backspace*)
`\a` suona il campanello della macchina
`\\` stampa il carattere `\`
`\"` stampa il carattere `"`

Le ultime due sequenze meritano un commento. Normalmente i doppi apici chiudono la descrizione del formato di una `printf`, perciò se si desidera visualizzare il carattere `"` lo si deve far precedere da `\`; una considerazione analoga vale per lo stesso carattere `\`.

È possibile inserire nella `printf`, al posto delle variabili, delle espressioni, di tipo specificato dal formato:

```
printf("Area: %d", 10*13);
```

Il `%d` ci indica che il risultato dell'espressione è di tipo intero; l'istruzione stamperà 130. Un'espressione può naturalmente contenere delle variabili:

```
printf("Area: %d", base*altezza);
```

Si può definire all'interno di una istruzione `printf` anche il numero di caratteri riservati per la visualizzazione di un valore, nel seguente modo:

```
printf("%5d%5d%5d", base, altezza, area);
```

Il `%5d` indica che verrà riservato un campo di cinque caratteri per la visualizzazione del corrispondente valore, che sarà sistemato a cominciare dall'estrema destra di ogni campo:



Se vengono inseriti degli spazi o altri caratteri nel formato, oltre alle descrizioni `%5d`, essi appariranno nelle posizioni corrispondenti. Inserendo poi un carattere − dopo il simbolo di percentuale e prima della lunghezza del campo il valore viene sistemato a cominciare dall'estrema sinistra della maschera. L'istruzione

```
printf("%-5d%-5d%5d", base, altezza, area);
```

visualizza dunque



1.5 Funzioni

Una *funzione* è costituita da un insieme di istruzioni che realizzano un compito: a partire da uno o più valori presi in input, essa restituisce un determinato valore in output.
Più avanti impareremo come creare nostre funzioni; esistono però delle funzioni predefinite o standard, già pronte all'uso, che il linguaggio mette a disposizione del programmatore. Da questo punto di vista non interessa come il compito affidato alla funzione venga svolto, basta sapere cosa deve esserle passato in entrata e cosa restituisce in uscita.

Un esempio di funzione C predefinita è `abs(i)`, che prende il nome da *absolute*: se dopo la parola `abs()`, all'interno delle parentesi tonde, viene inserito un numero intero, la funzione `abs()` ne restituisce il valore assoluto, che è quindi possibile catturare assegnandolo a una variabile o utilizzandolo direttamente all'interno di un'espressione. Se quindi `w` e `j` sono variabili di tipo intero, l'istruzione

```
w = abs(j);
```

assegna a `w` il valore assoluto di `j`. All'interno delle parentesi tonde può essere inserito direttamente un valore, come nel caso

```
w = abs(3);
```

che assegna a `w` il valore 3, o come nel caso

```
w = abs(-186);
```

che assegna a `w` il valore 186. In questo contesto, ribadiamo, la nostra attenzione non è rivolta al modo in cui viene svolto un certo compito ma a cosa immettere come argomento della funzione predefinita per ottenere un certo risultato. Naturalmente in qualche luogo è (pre)definito l'insieme di istruzioni che la compongono; nel caso della funzione `abs()` tale luogo è la libreria standard `math.h`. Perciò, per poter utilizzare tale funzione si deve dichiarare esplicitamente nel programma, prima del `main`, l'inclusione del riferimento a tale libreria.

```
#include <math.h>
```

Osserviamo nel Listato 1.6 un programma completo che utilizza la funzione `abs()`. Esso permette di risolvere il problema del calcolo della lunghezza di un segmento, i cui estremi vengono immessi dall'utente. Se consideriamo la retta dei numeri interi, ognuno dei due estremi può essere sia positivo sia negativo, per cui la lunghezza del segmento è pari al valore assoluto della differenza tra i due valori.

```
/* Esempio utilizzo di abs() */

#include <stdio.h>
#include <math.h>

main()
{
  int a, b, segmento, lunghezza;

  printf("\n\nLUNGHEZZA SEGMENTO\n");
  printf("Primo estremo: ");
  scanf("%d", &a);
  printf("Secondo estremo: ");
  scanf("%d", &b);

  segmento = a-b;
  lunghezza = abs(segmento);

  printf("Lunghezza segmento: %d\n", lunghezza);
}
```

Listato 1.6 Esempio di utilizzo di una funzione predefinita

L'esecuzione del programma del Listato 1.6, nel caso l'utente inserisca i valori 7 e –2, produrrà la seguente visualizzazione:

```
LUNGHEZZA SEGMENTO
Primo estremo: 7
Secondo estremo: -2
Lunghezza segmento: 9
```

Abbiamo detto che il risultato restituito da una funzione può essere inserito all'interno di un'espressione; ecco un esempio, in cui `j` ha valore 100 e `k` ha valore –73:

```
w = j*abs(k);
```

L'espressione precedente assegna a `w` il valore 7300, mentre `w = j*k` gli avrebbe assegnato –7300.

Anche `printf` e `scanf` sono funzioni standard C, alle quali si accede mediante `stdio.h`. Per questa ragione all'inizio del programma precedente, così come degli altri, abbiamo incluso il riferimento a tale libreria.

Dal punto di vista del programmatore, quello che interessa per ottenere un certo risultato è sapere:

1. 1. che esiste la funzione corrispondente;
2. 2. di quali informazioni essa ha bisogno;
3. 3. in quale librerie è contenuta.

Le funzioni standard sono catalogate rispetto all'applicazione cui sono dedicate; per esempio:

`stdio.h`	funzioni di input/output
`math.h`	funzioni matematiche
`string.h`	funzioni che operano su stringhe

Esistono molte librerie, ognuna delle quali contiene un certo numero di funzioni. Il programmatore può creare delle proprie funzioni ed eventualmente inserirle in file che diventano le sue librerie personali; quando lo desidera può includere nel programma tali librerie così come fa con quelle standard.

1.6 Fasi di programmazione

Qualsiasi versione del C si abbia a disposizione, e qualsiasi sistema operativo si impieghi, le fasi del lavoro del programmatore sono costituite da:

- • editing del programma;
- • precompilazione;
- • compilazione;
- • traduzione in codice oggetto;
- • *link*;
- • esecuzione.

I vari ambienti di programmazione si differenziano per gli strumenti che mettono a disposizione per tali fasi. Per la prima di esse si potrà "minutare" il programma con l'editor preferito, l'importante è che il file prodotto sia privo di quei caratteri speciali che vengono inseriti per la formattazione del testo e per altre funzioni. Nei sistemi più diffusi questo file è in formato ASCII.

La fase di precompilazione viene eseguita dal *preprocessore C* (vedi Capitolo 8), che ha il compito di espandere alcune forme abbreviate. È, per esempio, il preprocessore che si preoccupa di sostituire nel programma ai nomi delle costanti i loro valori, specificati con la macroistruzione `define` che abbiamo introdotto in questo capitolo. L'uscita del preprocessore, costituita dal codice sorgente espanso, viene elaborata dal *compilatore C* vero e proprio, che ricerca gli errori eventualmente presenti e traduce tale codice in istruzioni scritte nel linguaggio assemblatore. Questa versione del programma originario in linguaggio assemblatore viene passata all'*assembler*, che effettua la traduzione in una forma chiamata *codice oggetto rilocabile*. Questa forma non è ancora eseguibile dal sistema di elaborazione, perché deve

essere collegata alle librerie alle quali si fa riferimento negli `include`. Infatti il compilatore lascia in sospeso tutte le funzioni che vengono invocate nel programma ma che non vi sono definite; è il caso di `printf()` e `scanf()` che abbiamo già utilizzato. Il linker ricerca tali funzioni nelle librerie indicate: se le trova le collega, altrimenti restituisce dei messaggi di errore.

La precompilazione, la compilazione, l'assemblaggio e il link possono venire effettuati dal prompt di sistema richiamando un comando specifico, che spesso è `cc` seguito dal nome del file o dei file contenenti il testo del programma:

```
$ cc rettang.c
```

Se il comando non rileva errori riappare il prompt, altrimenti scorre sul video una lista degli errori. Il codice oggetto è adesso nel file `a.out` e si può mandare in esecuzione semplicemente digitandone il nome:

```
$ a.out
AREA RETTANGOLO

Valore base: 10
Valore altezza: 13
Base: 10
Altezza: 13
Area: 130
$
```

Se successivamente viene eseguita la compilazione di un altro programma, il nuovo codice oggetto rimpiazzerà il primo in `a.out`, per cui è bene ogni volta effettuare una copia di `a.out` su un diverso eseguibile. Il comando `cc` ha moltissime opzioni; una di esse, `-o`, permette di specificare direttamente il nome del file oggetto:

```
$ cc rettang.c -o rettang.exe
```

È poi possibile effettuare il link separatamente, il che consente, come vedremo in seguito, una notevole flessibilità nella programmazione.

Se si è in un ambiente dedicato, per sviluppare le varie fasi naturalmente basterà scegliere le opzioni relative dai menu messi a disposizione. Per esempio, con "Microsoft Visual C++", una volta editato il programma possiamo scegliere in sequenza le opzioni `Compile`, `Build` ed `Execute` dal menu `Build`.

1.7 Esercizi

1. Predisporre un programma che, utilizzando una sola istruzione `printf`, visualizzi:

```
Prove
 Tecniche di
  visualizzazione
```

2. Codificare un programma che calcoli la seguente espressione: `y=xa+b`, dove `x` è uguale a 5, `a` è uguale a 18 e `b` è uguale a 7; `x`, `a`, e `b` devono essere dichiarate come variabili intere. Si visualizzi infine il valore finale:

```
y = 97
```

3. Trasformare il programma dell'esercizio precedente in modo che il valore di `x` venga richiesto all'utente in fase di esecuzione.

4. Modificare il programma dell'esercizio precedente in modo che utilizzi le costanti `A` e `B` invece delle variabili `a` e `b`.

* 5. Scrivere un programma che calcoli e visualizzi le seguenti espressioni:

```
a = ZERO - abs(x)
b = TOP - abs(y)
c = a*b
```

dove `x` e `y` sono variabili intere immesse dall'utente, `ZERO` e `TOP` sono costanti intere di valore 0 e 1000.

6. Predisporre un programma che mostri chiaramente le diverse funzionalità delle sequenze di escape all'interno delle istruzioni `printf`.

7. Determinare il valore assunto dalle variabili `a`, `b` e `c` al termine della successiva sequenza di istruzioni:

```
a = -2;
b = a+1;
b = b - abs(a);
c = a*b;
b = 3;
```

8. Indicare tutti gli errori commessi nel seguente listato.

```
#include <stdio.h>

/* Tutto Sbagliato!!! */

#define BASE 3
#define ALTEZZA

main()
  area int;

  area = BASE x ALTEZZA;
  printf("Base: d\n", BASE);
  printf("Altezza: %d\n", ALTEZZA)
  printf("Area: %d\n"; area);
}
```

2.1 L'istruzione

Quando si desidera eseguire un'istruzione al verificarsi di una certa condizione, si utilizza l'istruzione `if`. Per esempio, se si vuole visualizzare il messaggio `minore di 100` solamente nel caso in cui il valore della variabile intera `i` è minore di 100, si scrive

```
if(i<100) printf("minore di 100");
```

La sintassi dell'istruzione `if` è

```
if(espressione) istruzione
```

dove la valutazione di *`espressione`* controlla l'esecuzione di *`istruzione`*: se *`espressione`* è vera viene eseguita *`istruzione`*. L'espressione `i<100` è la condizione logica che controlla l'istruzione di stampa e pertanto la sua valutazione potrà restituire soltanto uno dei due valori booleani vero o falso, che in C corrispondono rispettivamente ai valori interi uno e zero. In C, a differenza che in altri linguaggi come il Pascal, non esistono variabili logiche, per cui *falso* è lo zero e *vero* è ciascun valore diverso da zero (in effetti uno è diverso da zero!). Quindi `i<100` è anche un'espressione intera la cui valutazione restituisce 1 quando la variabile `i` è minore di 100 e 0 quando `i` è maggiore o uguale a 100. L'esempio precedente è funzionalmente identico alla successiva sequenza, dove `a` è ancora una variabile intera:

```
a = i<100;
```

```
if(a!=0)
  printf("minore di 100");
```

L'assegnamento `a=i<100` è del tutto lecito, perché viene valutata l'espressione logica `i<100`, che può restituire 1 (vero) o 0 (falso). Il risultato è dunque un numero intero, che viene assegnato alla variabile `a`.

Se osserviamo la sintassi del costrutto `if`, notiamo che nel primo esempio *`espressione`* è `i<100` mentre nel secondo è `a!=0`, e in entrambi *`istruzione`* è `printf()`. L'operatore `!=` corrisponde a *diverso da*, per cui valutare l'espressione `a!=0` significa chiedersi se il valore di a è diverso da zero. Ma questo equivale a chiedersi se il valore di `a` è vero, che è un controllo eseguito per default; avremmo quindi potuto scrivere anche semplicemente:

```
a = i<100;
if(a)
  printf("minore di 100");
```

Nei paragrafi successivi esamineremo tutti gli operatori di confronto, riprenderemo il discorso sulle espressioni aritmetiche e logiche e vedremo come si integrano le une nelle altre.

La sintassi completa dell'istruzione `if` è:

`if(`*`espressione`*`)` *`istruzione1`* `[else` *`istruzione2`*`]`

dove la valutazione di *`espressione`* controlla l'esecuzione di *`istruzione1`* e quella di *`istruzione2`*: se *`espressione`* è vera viene eseguita *`istruzione1`*, se è falsa viene eseguita *`istruzione2`*. .
Nell'esempio precedente è stato omesso il ramo `else`; il fatto è del tutto legittimo poiché tale ramo è opzionale, come evidenziato dalle parentesi quadre della forma sintattica completa e come vedremo anche nel prossimo paragrafo. Modifichiamo ora l'esempio in modo da visualizzare un messaggio anche quando la variabile `i` non è minore di 100 (Listato 2.1).

```
/* Utilizzo if-else */

#include <stdio.h>

main()
{
int i;

printf("Dammi un intero: ");
scanf("%d", &i);

if(i<100)
  printf("minore di 100\n");
else
  printf("maggiore o uguale a 100\n");
}
```

Listato 2.1 Esempio di diramazione del flusso di esecuzione

Abbiamo introdotto il ramo `else` dell'istruzione `if`, inserendo l'istruzione che visualizza `maggiore o uguale a 100`, da eseguire nel caso l'espressione di controllo risulti falsa. Si noti che i rami `if-else` si escludono mutuamente, o viene eseguita la prima `printf` o viene eseguita la seconda, mai entrambe.

2.2 Le istruzioni composte

Nell'istruzione `if` soltanto un'istruzione semplice viene controllata dalla valutazione di *`espressione`*. Scrivendo

```
if(i<100)
  printf("minore di 100 ");
printf("istruzione successiva");
```

la seconda `printf` viene infatti eseguita indipendentemente dal risultato della valutazione di `i<100`. Ciò non ha niente a che vedere, ovviamente, con l'aspetto grafico della sequenza: lo stesso risultato si ottiene scrivendo

```
if(i<100)
  printf("minore di 100 ");
  printf("istruzione successiva");
```

Lo stesso dicasi per la clausola `else`: nel frammento di codice

```
if(i<100)
  printf("minore di 100 ");
else
  printf("maggiore o uguale a 100 ");
printf("istruzione successiva alla clausola else");
```

l'ultima `printf` viene eseguita indipendentemente dal fatto che `i` risulti minore oppure maggiore o uguale a 100.

Supponiamo ora di voler aggiungere nel Listato 2.1 un'ulteriore istruzione, oltre alla `printf`, ai due rami del costrutto `if-else`. Inizializziamo due variabili intere, `mag_100` e `min_100`, a zero. Nel caso in cui `i` risulti essere minore di 100, assegniamo il valore 1 a `min_100`, altrimenti lo assegniamo a `mag_100`. In termini tecnici diciamo che alziamo il *flag* `mag_100` o il *flag* `min_100` in base al risultato del controllo effettuato dall'istruzione `if` (vedi Listato 2.2).

```
/* Esempio istruzioni composte */

#include <stdio.h>

int i;
int mag_100;
int min_100;

main()
{
mag_100 = 0;
min_100 = 0;

printf("Dammi un intero: ");
scanf("%d", &i);

if(i<100) {
  printf("minore di 100\n");
  min_100 = 1;
}
else {
  printf("maggiore o uguale a 100\n");
  mag_100 = 1;
}
}
```

Listato 2.2 Esempio di utilizzo di istruzioni composte

A tale scopo utilizziamo *l'istruzione composta*, detta anche *blocco*, costituita da un insieme di istruzioni inserite tra parentesi graffe che il compilatore tratta come un'istruzione unica. Quindi la scrittura

```
{
```

```
  printf("minore di 100\n");
  min_100 = 1;
}
```

è un'istruzione composta costituita da due istruzioni. Nel listato completo la parentesi graffa aperta è stata inserita nella stessa linea dell'istruzione `if`, dopo la chiusura della parentesi tonda; ovviamente il significato non cambia: l'importante è saper riconoscere l'inizio e la fine dell'istruzione composta. Analoga considerazione vale per la clausola `else`.

Se nella sintassi assumiamo che un'istruzione possa essere semplice o composta, l'esempio del paragrafo precedente e quello appena visto sono riconducibili alla stessa forma del comando:

```
if(espressione) istruzione1 [else istruzione2]
```

dove per entrambi gli esempi *`espressione`* corrisponde a `i<100`, mentre *`istruzione1`* e *`istruzione2`* corrispondono rispettivamente a:

Istruzione semplice	*Istruzione composta*

`printf("minore di 100\n");`	`{`
	`printf("minore di 100\n");`
	`min_100 = 1;`
	`}`

`printf("maggiore o uguale`
`a 100\n");`
	`{`
	`printf("maggiore o uguale a 100\n");`
	`mag_100 = 1;`
	`}`

Se non si fossero utilizzate le parentesi graffe il significato del programma sarebbe stato ben diverso:

```
if(i<100)
  printf("minore di 100\n");
  min_100 = 1;
else
  printf("maggiore o uguale a 100\n");
  mag_100 = 1;
```

Ciò che può fuorviare è l'aspetto grafico del codice, ma dato che l'indentazione non viene considerata la compilazione rivelerà un errore trovando un'istruzione `else` spuria, cioè non ricollegabile a un'istruzione `if`. Non essendo stata aperta la parentesi graffa, l'`if` regge la sola istruzione `printf("è minore di 100\n")`, dopo di che, se non trova la clausola `else`, si chiude.

Un'istruzione composta può essere immessa nel programma dovunque possa comparire un'istruzione semplice e quindi indipendentemente dal costrutto `if-else`; al suo interno, dopo la parentesi graffa aperta e prima delle altre istruzioni, possono essere inserite delle dichiarazioni di *variabili locali* a quel blocco, variabili – cioè – che possono essere utilizzate fino alla chiusura del blocco stesso:

```
{
  dichiarazione di variabili
  istruzione1
  istruzione2
  ...
  istruzioneN
}
```

2.3 `if` annidati

Il costrutto `if` è un'istruzione e quindi può comparire all'interno di un altro `if`, come si deduce dalla sintassi generale, nel ruolo di *`istruzione`*. Quando ciò si verifica si parla di if annidati. Nell'esempio:

```
if(i<100)
  if(i>0)
    printf("minore di 100 e maggiore di zero");
```

il secondo controllo (`i>0`) viene effettuato soltanto se il primo (`i<100`) ha dato esito positivo. Se anche il secondo `if` risulta vero si visualizza il messaggio. Si osservi che, dopo il primo `if`, non è necessario inserire il secondo `if` e la `printf` all'interno di parentesi graffe, in quanto queste costituiscono già un'unica istruzione semplice.

Aggiungiamo ora al nostro esempio un ramo `else`:

```
if(i<100)
  if(i>0)
    printf("minore di 100 e maggiore di zero");
  else
    printf("minore di 100 ma non maggiore di zero");
```

Come fa capire il messaggio della seconda `printf`, l'`else` che abbiamo aggiunto si riferisce al secondo `if`, cioè a quello più interno, il quale insieme alle due `printf` e all'`else` costituiscono ancora un'unica istruzione. Per fare in modo che l'`else` si riferisca al primo `if` bisogna usare le parentesi graffe:

```
if(i<100) {
  if(i>0)
    printf("minore di 100 e maggiore di zero");
}
else
  printf("maggiore o uguale a 100");
```

Se invece avessimo anche l'`else` dell'`if` più interno le parentesi graffe sarebbero superflue, e si potrebbe scrivere:

```
if(i<100)
  if(i>0)
    printf("minore di 100 e maggiore di zero");
  else
    printf("minore di 100 ma non maggiore di zero");
else
  printf("maggiore o uguale a 100");
```

Ciò è reso possibile dal fatto che il tutto viene considerato come un'unica istruzione. Modifichiamo ora il nostro esempio in modo che vengano visualizzati due messaggi diversi a seconda che la variabile `i` sia maggiore oppure uguale a 100, e non soltanto un messaggio generico come sopra:

```
if(i<100)
  if(i>0)
    printf("minore di 100 e maggiore di zero");
  else
    printf("minore di 100 ma non maggiore di zero");
else
  if(i==100)
    printf("uguale a 100");
  else
    printf("maggiore di 100");
```

Come si può osservare, anche la parte `istruzione` dell'`else` può essere un'istruzione `if`, la quale, a sua volta, può avere un proprio ramo `else`. L'operatore == ha il significato di *uguale a*, risponde vero se l'operando che lo precede e quello che lo segue sono uguali, falso altrimenti.

Un'ulteriore modifica del nostro esempio consiste nel dare più informazioni nel caso che `i` sia minore di 100:

```
if(i<100)
  if(i>0)
    printf("minore di 100 e maggiore di zero");
```

```
4     else
5       if(i==0)
6         printf("uguale a zero");
7       else
8         printf("minore di zero");
9   else
10    if(i==100)
11      printf("uguale a 100");
12    else
13      printf("maggiore di 100");
```

È importante notare che nell'esempio precedente non sono richieste parentesi graffe; ciò in virtù del fatto che:

- le righe `1..13` sono un'unica istruzione `if-else` la quale ha per *`istruzione1`* le `righe 2..8` e per *`istruzione2`* le righe `10..13`;
- le righe `2..8` sono un'unica istruzione `if-else` la quale ha per *`istruzione1`* la riga `3` e per *`istruzione2`* le righe `5..8`;
- le righe `5..8` sono un'unica istruzione `if-else` la quale ha per *`istruzione1`* la riga `6` e per *`istruzione2`* la riga `8`;
- le righe `10..13` sono un'unica istruzione `if-else` la quale ha per *`istruzione1`* la riga `11` e per *`istruzione2`* la riga *13*.

✓ **NOTA**

Quanto sopra può essere scritto in modo più compatto:

```
if(i<100)
  if(i>0)
    printf("minore di 100 e maggiore di zero");
  else if(i==0)
    printf("uguale a zero");
  else
    printf("minore di zero");
else if(i==100)
  printf("uguale a 100");
else
  printf("maggiore di 100");
```

Questo schema è frequente nei programmi C ed è molto comodo per simulare l'istruzione *`elseif`*, tipica di altri linguaggi.

2.4 Espressioni

2.4.1 Espressioni condizionali

Si definisce *espressione aritmetica* un insieme di variabili, costanti e richiami di funzione connessi da operatori aritmetici. Il risultato di un'espressione aritmetica è sempre un valore numerico. La Figura 2.1 mostra gli operatori aritmetici e le loro priorità in ordine dalla più alta alla più bassa. Da osservare come non sia presente l'operatore di elevamento a potenza.

```
negazione (-unario)
 moltiplicazione (*), divisione (/), modulo (%)
somma (+), sottrazione (-)
= (assegnamento)
```

Figura 2.1 Gerarchia degli operatori aritmetici e di assegnamento

Quello di negazione è l'unico operatore unario, cioè che si applica a un solo operando. Se `x` ha valore 5, l'espressione

```
-x;
```

restituisce –5, mentre

```
2 * -(x-6);
```

restituisce 2. Si noti che mentre il `-` anteposto alla parentesi tonda aperta corrisponde all'operatore unario di negazione, l'altro rappresenta l'operatore binario di sottrazione.

L'operatore di modulo, `%`, consente di ottenere il resto della divisione intera tra l'operando che lo precede e quello che lo segue. Quindi, sempre nell'ipotesi che `x` valga 5,

```
34 % x;
```

ha valore 4, perché (34 : 5 = 6 con resto 4).

All'interno delle espressioni aritmetiche, la priorità degli operatori segue le regole dell'algebra. La valutazione di una espressione contenente operazioni matematiche avviene esaminandola da sinistra a destra più volte, dunque gli operatori sono associativi da sinistra verso destra. Tutte le operazioni di negazione sono eseguite per prime, quindi l'espressione è esaminata nuovamente per eseguire tutte le moltiplicazioni, divisioni e le operazioni modulo. Infine l'espressione viene sottoposta a scansione ancora una volta per eseguire le addizioni e le sottrazioni. La priorità degli operatori può essere alterata mediante le parentesi tonde: vengono valutate per prime le operazioni all'interno delle parentesi tonde più interne.

Osserviamo le seguenti espressioni, nell'ipotesi che le variabili intere `a`, `b` e `c` abbiano rispettivamente valore: 7, 3 e 5. Il risultato di

```
a + b - 15 + 4 * c
```

è 15, mentre il risultato di

```
a + b + c % 2
```

è 11, in quanto l'operatore modulo restituisce il resto della divisione intera tra il valore di `c` (5) e 2, ossia 1. Il risultato di

```
(a+b) * 32 + 4 * c
```

è 340 mentre quello di

```
((((((c + 6) * 3 + a) / 2) + 10 * 4) / 12) + b)
```

è 8. Non esiste alcun limite al numero di coppie di parentesi tonde impiegate.

L'assegnamento = è anch'esso un operatore, e ha quindi la sua posizione all'interno della scala di priorità (Figura 2.1); la sua priorità è minore di quella di tutti gli altri; per questa ragione nell'espressione

```
y = z * 2 / x
```

prima viene valutata completamente la parte a destra dell'assegnamento e poi il risultato viene immesso nella variabile `y`. L'operazione di assegnamento può essere multipla, per esempio:

```
x = y = z
```

In questo caso il valore di `z` viene assegnato a `y` e a `x`. Analogamente si può avere

```
a = b = c = f = d
```

dove il valore di `d` viene assegnato ad `a`, `b`, `c` e `f`, o anche

```
x = y = t = z*2/x;
```

dove il valore restituito da `z*2/x` viene assegnato a `x`, `y` e `t`.

2.4.2 Espressioni logiche

Un'*espressione logica* è un'espressione che genera come risultato un valore vero o falso (abbiamo visto che in C non esiste il tipo booleano, presente in altri linguaggi), e viene utilizzata dalle istruzioni di controllo. Le espressioni logiche, per la precisione, producono come risultato 1 per vero e 0 per falso (qualsiasi valore numerico diverso da zero viene comunque considerato vero). Un semplice esempio di espressione logica è una variabile il cui contenuto può essere interpretato in due modi: vero se diverso da zero, falso se uguale a zero.

Le espressioni logiche possono contenere gli *operatori relazionali*, usati per confrontare fra loro dei valori, riportati in Figura 2.2.

```
> (maggiore di)  >= (maggiore uguale)
< (minore di)    <= minore uguale)

== (uguaglianza) != (disuguaglianza)
```

Figura 2.2 Gerarchia degli operatori relazionali

Si noti come l'operatore di uguaglianza == sia diverso anche nella notazione da quello di assegnamento =, fatto a nostro avviso positivo e non comune negli altri linguaggi di programmazione.

La priorità di >, >=, <, e <= è la stessa ed è maggiore di == e `!=`. Dunque in

```
x>y == z>t
```

viene valutato prima `x>y` e `z>t` e successivamente verificata l'uguaglianza tra i due risultati come se l'espressione fosse: `(x>y)==(z>t)`.

```
!  (NOT logico)

&& (AND logico)

|| (OR logico)
```

Figura 2.3 Gerarchia degli operatori logici

Gli *operatori logici* consentono invece di concatenare fra di loro più espressioni logiche e di negare il risultato di un'espressione logica; essi hanno la scala di priorità di Figura 2.3 e la seguente *tavola di verità*, dove `0` corrisponde a falso e `1` a vero.

x	**y**	**x&&y**	**x||y**	**!x**
0	0	0	0	1

0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Il connettivo `&&` restituisce vero se e solo se il risultato di entrambe le due operazioni logiche, quella che lo precede e quella che lo segue, risultano vere. L'espressione

```
x==y && a>b
```

è quindi vera se `x` è uguale a `y` e contemporaneamente `a` è maggiore di `b`.

Il connettivo `||` restituisce vero se almeno una delle espressioni logiche che esso concatena risulta vera. Quindi l'espressione

```
b<c || t!=r
```

restituisce falso solamente se `b` non è minore di `c` e contemporaneamente `t` è uguale a `r`.

L'espressione

```
!y
```

restituisce vero se `y` è falso e viceversa. L'operatore `!` è di tipo unario, gli altri sono binari.

L'ordine di priorità complessiva degli operatori logici e relazionali è mostrato in Figura 2.4. Naturalmente, si possono utilizzare le parentesi tonde per alterare la priorità; per esempio, in

```
(nome1!=nome2 || cogno1>cogno2) && presenti < 524
```

viene valutato prima `||` di `&&`.

Il risultato di un'espressione logica può essere assegnato a una variabile, come abbiamo già visto tra gli esempi del primo paragrafo:

```
a = i<100
```

Dalla tabella gerarchica degli operatori deduciamo che l'assegnamento è quello con minor priorità; è per questa ragione che viene valutato innanzitutto `i<100`; se `i` risulta minore di 100 `a` assume valore 1, altrimenti assume valore 0. L'aggiunta di un punto e virgola trasforma l'espressione in una istruzione:

```
a = i<100;
```

Esaminiamo ora un altro esempio:

```
y = a==b;
```

dove `y` assume valore 1 se `a` è uguale a `b`, 0 viceversa. Dunque è lecito anche questo assegnamento, la cui interpretazione lasciamo al lettore:

```
x = (x==y) && (z!=t || m>=n);
```

Dato che le espressioni logiche restituiscono un risultato numerico, non esistono differenze tra le espressioni logiche e quelle aritmetiche; la Figura 2.4 riporta la scala di priorità complessiva degli operatori che abbiamo presentato in questo capitolo.

```
!      - (unario)
```

`*`	`/`	`%`	
`+`	`-`		
`>`	`>=`	`<`	`<=`
`==`	`!=`		
`&&`			
`||`			
`?:`			
`=`			

Figura 2.4 Gerarchia degli operatori esaminati in questo capitolo

Un'espressione può contenere una combinazione di operatori aritmetici, logici e relazionali. L'espressione

```
x + sereno + (i<100)
```

restituisce la somma di `x`, di `sereno` e del risultato di `i<100`, che è 1 se `i` è minore di 100, zero altrimenti. Per esempio, se `x` vale 5, `sereno` 6 e `i` 98, l'espressione restituisce 12.

È un'espressione anche la seguente:

```
y = x + sereno + (i<100)
```

dove l'assegnamento a `y` viene effettuato alla fine, perché = ha la priorità più bassa. Essendo l'operatore di assegnamento trattato alla stregua degli altri, sarà lecita anche la seguente espressione:

```
i>n && (x=y)
```

che mette in AND le espressioni `i>n` e `x=y`. La prima è vera se il valore di `i` è maggiore di `n`. La seconda corrisponde all'assegnamento del valore di `x` alla variabile `y`; tale assegnamento viene effettuato all'interno dell'espressione, dopo di che se il valore di `x` è diverso da zero l'espressione risulta vera, altrimenti falsa.

✓ **NOTA**

Queste caratteristiche rendono il C un linguaggio flessibile che consente la scrittura di codice sintetico ed efficiente, ma anche difficilmente interpretabile. Prendiamo ad esempio il seguente frammento di codice:

```
if((x=y)  && i>n)
  printf("vero");
else
  printf("falso");
```

L'espressione presente nell'`if` contiene (nasconde!) un assegnamento. È chiaro che in questo modo risparmiamo una linea di codice ma, ci pare, a scapito della chiarezza. In particolare, si faccia attenzione a non confondere l'operatore relazionale == con quello di assegnamento. Dati i due casi:

```
if(x=y)                       if(x==y)
  printf("vero");               printf("vero");
else                          else
  printf("falso")               printf("falso");
```

in nessuno dei due il compilatore segnalerà un errore, poiché le espressioni presenti sono sintatticamente corrette. Nel caso di sinistra, nell'`if` il valore di `y` viene assegnato a `x` e se risulta essere diverso da zero viene stampato vero, altrimenti falso. Nel caso di destra non avviene nessun assegnamento: se `x` risulta uguale a `y` verrà stampato vero, altrimenti falso. L'effettuare un assegnamento non voluto è uno degli errori più frequenti e quindi vale la pena rimarcare un'ultima volta che l'operatore di confronto di uguaglianza è ==.

Nell'espressione

```
(x=y)  && i>n
```

non è stato necessario racchiudere `i>n` tra parentesi tonde perché l'operatore > ha una priorità maggiore di `&&`. A volte, in special modo nelle espressioni molto lunghe, può risultare difficile comprendere a prima vista, in fase di programmazione, l'ordine di valutazione, per cui niente ci vieta di aggiungere delle parentesi tonde per rafforzare il concetto che vogliamo esprimere:

```
(x=y) && (i>n)
```

Per esempio,

```
((x) && (a>b)  || ((c<=d) && (f!=r))
```

è certamente equivalente a

```
x && a>b || c<=d && f!=r
```

ma la prima forma ci sembra più leggibile e può facilitare la revisione del programma.

2.4.3 Espressioni condizionali

Una *espressione condizionale* si ottiene con l'operatore ternario `?:` che ha la seguente sintassi:

`espr1 ? espr2 : espr3`

Se la valutazione di *`espr1`* restituisce vero, il risultato è uguale a *`espr2`*, altrimenti è uguale a *`espr3`*. Per esempio,

```
x==y ? a : b
```

significa: "se `x` è uguale a `y`, allora `a`, altrimenti `b`".
Si può utilizzare l'operatore `?:` per assegnare un valore a una variabile, come nel caso che segue:

```
v=x==y ? a*c+5 : b-d;
```

Se `x` è uguale a `y`, a `v` viene assegnato il valore di `a*c+5`, altrimenti gli viene assegnato il valore di `b-d`. Le espressioni *`espr1`*, *`espr2`* ed *`espr3`* vengono valutate prima di `?:` e l'assegnamento viene effettuato dopo, data la posizione dell'operatore condizionale nella tavola gerarchica.
L'espressione condizionale può essere sempre sostituita da un `if` corrispondente: nel caso precedente, per esempio, avremmo potuto scrivere:

```
if(x==y)
  v = a*c+5;
else
  v = b-d;
```

Essendo quella condizionale un'espressione come tutte le altre, può essere inserita in qualsiasi posizione sia appunto lecito scrivere un'espressione. Potremmo cioè scrivere:

```
x = a*x+(x==z ? b : c)+d;
```

corrispondente a

```
if(x==z)
  x = a*x+b+d;
else
  x = a*x+c+d;
```

L'istruzione

```
printf("%d maggiore o uguale a %d", (a>b?a:b), (a<=b?a:b));
```

restituisce:

- • se `a` ha valore 5 e `b` ha valore 3:
  `5 maggiore o uguale a 3`
- • se `a` è uguale a 100 e `b` è uguale a 431:
  `431 maggiore o uguale a 100`
- • se `a` è uguale a 20 e `b` è uguale a 20:
  `20 maggiore o uguale a 20`

L'operatore `?:` può essere inserito in qualsiasi espressione, dunque anche all'interno della condizione che controlla l'`if`:

```
if((a>=b ? a : b) >= c )
  printf("nero");
else
  printf("bianco");
```

In generale, quando più operatori con la stessa priorità devono essere valutati in un'espressione, lo standard C non garantisce l'ordine di valutazione. Nell'espressione

```
x > y <= z
```

non possiamo sapere se verrà valutata prima la condizione `x > y` e successivamente se il risultato (zero o uno) è minore o uguale a `z`, o viceversa. In ogni caso è buona norma non scrivere codice dipendente dall'ordine di valutazione per non rischiare di produrre programmi non portabili tra versioni C differenti.

Esiste comunque un'eccezione. Per gli operatori `&&`, `||`, e `?:` il linguaggio garantisce la valutazione delle espressioni da sinistra verso destra. Per esempio, in

```
x==y && a>b
```

l'espressione `a>b` sarà valutata soltanto se il risultato dell'espressione `x==y` è vero. Analogamente in

```
b<c || t!=r
```

l'espressione `t!=r` sarà valutata soltanto se il risultato dell'espressione `b<c` è falso.

Nell'espressione

```
x>y ?  a>b ?  a : b : y
```

prima viene valutata `a>b`, quindi restituito `a` oppure `b`; successivamente è valutata `x>y`, dato che `?:` è associativo da destra verso sinistra.

2.5 Variabili carattere

Sino a questo momento abbiamo lavorato con variabili di tipo intero. Introduciamo ora variabili di tipo carattere, che scopriremo avere in comune con le prime molto di più di quanto potremmo supporre.

Le variabili di tipo carattere assumono valori alfanumerici che comprendono le lettere dell'alfabeto minuscole e maiuscole, le cifre decimali, la punteggiatura e altri simboli. Scrivendo

```
char x, y, z;
```

la parola chiave `char` specifica che gli identificatori `x`, `y` e `z` che la seguono si riferiscono a variabili di tipo carattere. La definizione fa sì che venga riservato uno spazio in memoria sufficiente per contenere un carattere alfanumerico. La dimensione può variare rispetto all'implementazione; molte versioni riservano per i `char` uno spazio di un byte, il che permette di fare riferimento a 256 configurazioni di bit distinti e quindi individuare ogni carattere del codice in uso sulla macchina. I codici di più ampio utilizzo sono l'ASCII e l'EBCDIC. Per assegnare un valore costante a una variabile `char` lo si deve racchiudere tra apici singoli:

```
x = 'A';
y = ';';
z = '&';
```

Come si può facilmente dedurre dal codice ASCII riportato in Appendice D, si hanno le seguenti corrispondenze.

Carattere	Decimale	Esadecimale	Binario
A	65	41	01000001
;	59	3B	00111011
&	38	26	00100110

A ogni carattere presente nel codice corrisponde una rappresentazione numerica univoca, per cui è possibile confrontare due simboli non solamente con uguaglianze e diseguaglianze, ma anche per mezzo di tutti gli altri operatori relazionali.

"`A`" (65) è maggiore di "`;`" (59) che a sua volta è maggiore di "`&`" (38). Osservando il codice ASCII possiamo vedere che le lettere alfabetiche maiuscole sono in ordine crescente da `A` (65) a `Z` (90), le minuscole vanno da `a` (98) a `z` (122) e le cifre decimali da `0` (48) a `9` (57). Dunque ha perfettamente senso l'istruzione condizionale

```
if(x=='A')
  printf("Si tratta di una A");
```

ma anche

```
if(x>='A' && x<='Z')
  printf("Si tratta di una lettera maiuscola");
```

Per poter visualizzare dei `char` con una `printf` si deve come al solito indicarne il formato; per esempio:

```
printf("%c %c %c", x, y, z);
```

Come abbiamo già osservato, i simboli di percentuale tra i doppi apici definiscono il formato di stampa delle corrispondenti variabili; questa volta le `c` (*character*) specificano che si tratta di caratteri. L'esecuzione dell'istruzione restituisce

```
A ; &
```

mentre l'istruzione

```
printf("%c%c%c", x, y, z);
```

restituisce

```
A;&
```

Naturalmente si possono stampare valori di tipi diversi all'interno della stessa `printf`, indicandone il formato nell'ordine corretto. Se `n` e `m` sono variabili `int` si può scrivere

```
x = 'a'; y = 'b';
```

```
n = 100; m = 4320;
printf("%c = %d   %c = %d", x, n, y ,m);
```

che restituisce

```
a = 100   b = 4320
```

Per mezzo dell'istruzione `scanf` si possono poi inserire caratteri a tempo di esecuzione:

```
scanf("%c", &x)
```

Il formato `%c` indica che si tratta di un carattere, così come `%d` indicherebbe che si tratta di un intero da visualizzare in formato decimale.

Esistono altre due funzioni standard di input/output, cui si può far riferimento per mezzo di `<stdio.h>`, che permettono di leggere e scrivere caratteri: `getchar` e `putchar`.

Se `x` è una variabile di tipo carattere,

```
x = getchar();
```

bloccherà il programma in attesa di un carattere introdotto da tastiera. Si noti che la presenza delle parentesi tonde è necessaria anche se dentro non vi è racchiuso alcun argomento.

Per visualizzare un carattere abbiamo invece la funzione

```
putchar(x);
```

Vediamo una semplice applicazione di queste due funzioni. I due programmi del Listato 2.3 hanno funzionamenti identici: i caratteri dovranno essere digitati uno dietro l'altro e successivamente dovrà essere battuto un Invio.

✓ **NOTA**

Se il programma dovesse prevedere l'immissione di più valori in tempi diversi, l'inserimento di un carattere potrebbe costituire un problema, dato che la digitazione del tasto di Invio da tastiera corrisponde a un carattere accettato da `scanf("%c",…)`. In tali casi verrà utilizzata un'opportuna ulteriore lettura di un carattere in una variabile ausiliaria tramite un'istruzione del tipo `scanf("%c", pausa)` o `pausa=getchar()`.

```
#include <stdio.h>

main()
{
char x, y, z;
printf("digita tre carat.: ");
scanf("%c%c%c", &x, &y, &z);


printf("Hai digitato: ");
printf("%c%c%c\n", x, y, z);




}
```

```
#include <stdio.h>

main()
{
char x, y, z;
printf("digita tre carat.: ");
x = getchar();
y = getchar();
z = getchar();
printf("Hai digitato: ");
putchar(x);
putchar(y);
putchar(z);
putchar('\n');
 }
```

Listato 2.3 Due esempi di input/output di caratteri

2.6 Istruzione `switch-case`

Le decisioni a più vie possono essere risolte utilizzando più `if-else` in cascata:

```
if(espressione1)
  istruzione1
else
  if(espressione2)
    istruzione2
  else
    if(espressione3)
      istruzione3
    ...
        else
          istruzioneN
```

Ognuna delle istruzioni può essere formata da più istruzioni, se racchiuse tra parentesi graffe (istruzioni composte).

Un'altra soluzione è data dal costrutto `switch-case`, che consente di implementare decisioni multiple basandosi sul confronto fra il risultato di un'espressione (`int` o `char`) e un insieme di valori costanti (Figura 2.5).

```
switch(espressione) {
  case costante1:
     istruzione
     ...
  case costante2:
     istruzione
     ...
  case costante3:
     istruzione
     ...
  [default:
     istruzione
     ...       ]
}
```

```
switch(espressione) {
  case costante1:
     istruzione
     ...
     break;
  case costante2:
     istruzione
     ...
     break;
  case costante3;
     istruzione
     ...
     break;
  ...
  [default:
     istruzione
     ...       ]
}
```

Figura 2.5 A sinistra sintassi del costrutto `switch-case`; a destra forma spesso utilizzata del costrutto `switch-case`

La parola `switch` è seguita da una *`espressione`*, racchiusa tra parentesi tonde, il cui risultato deve essere di tipo `int` o `char`. Il resto del costrutto è formato da un'istruzione composta, costituita da un numero qualsiasi di sottoparti, ciascuna delle quali inizia con la parola chiave `case`, seguita da un'espressione costante intera o carattere. Questa è separata, tramite un simbolo di due punti, da una o più istruzioni.

In fase di esecuzione, viene valutata *`espressione`* e il risultato viene confrontato con *`costante1`*: se i due valori sono uguali il controllo passa alla prima istruzione che segue i due punti corrispondenti, altrimenti si prosegue confrontando il risultato dell'espressione con *`costante2`*, e così di seguito. Una volta che il controllo è trasferito a una certa istruzione vengono eseguite linearmente tutte le rimanenti istruzioni presenti nello `switch-case` a sinistra della Figura 2.5.

Spesso, nell'utilizzo di questo costrutto, il programmatore desidera che vengano eseguite solamente le istruzioni associate a un singolo `case`. A questo scopo abbiamo inserito in Figura 2.5 a destra, al termine di ogni `case`, l'istruzione `break`, che causa l'uscita immediata dallo `switch`. Si osservi comunque che anche la situazione a sinistra può rivelarsi utile in particolari circostanze e va interpretata correttamente come una possibilità in più offerta dal linguaggio.

Se l'espressione non corrisponde a nessuna delle costanti, il controllo del programma è trasferito alla prima istruzione che segue la parola riservata `default` (se presente).

I valori *`costante1`*, *`costante2`*, .., *`costanteN`* possono essere delle espressioni costanti come `3*2+5` o `5*DELTA`, dove `DELTA` è una costante. Il Listato 2.4 è un esempio di utilizzo del costrutto `switch-case`.

```
/* Esempio utilizzo case */

#include <stdio.h>

int x;
main()
{
printf("Digita una cifra: ");
scanf("%d", &x);

switch(x) {
  case 0:
    printf("zero\n");
    break;
  case 1:
    printf("uno\n");
    break;
  case 2:
    printf("due\n");
    break;
  case 3:
    printf("tre\n");
    break;
  case 4:
    printf("quattro\n");
    break;
  case 5:
    printf("cinque\n");
    break;
  default:
    printf("non compreso\n");
    break;
}
}
```

Listato 2.4 Esempio di diramazione multipla del flusso di esecuzione

È possibile far corrispondere a un gruppo di istruzioni più costanti, ripetendo più volte la parola chiave `case` seguita dai due punti, come nel Listato 2.5.

```
/* Esempio utilizzo case */

#include <stdio.h>
```

```
char x;
main()
{
printf("Digita una cifra: ");
scanf("%c", &x);

switch(x) {
  case '2':
  case '4':
  case '6':
    printf("pari\n");
    break;
  case '1':
  case '3':
  case '5':
    printf("dispari\n");
    break;
  default:
    printf("altro\n");
}
}
```
Listato 2.5 Più valori costanti corrispondono allo stesso gruppo di istruzioni

Esercizi

1. Scrivere un programma che richieda in ingresso tre valori interi distinti e ne determini il maggiore.

* 2. Ripetere l'Esercizio 1 ma con quattro valori in ingresso.

3. Ripetere l'Esercizio 2 nell'ipotesi che i quattro valori possano anche essere tutti uguali, caso nel quale il messaggio da visualizzare dev'essere `Valori identici`.

* 4. Ripetere l'Esercizio 1 ma individuando anche il minore dei tre numeri in input.

* 5. Se le variabili intere `a`, `b` e `c` hanno rispettivamente valore 5, 35 e 7, quali valore viene assegnato alla variabile `ris` dalle seguenti espressioni?

```
1)   ris = a+b*c
2)   ris = (a>b)
3)   ris = (a+b) * (a<b)
4)   ris = (a+b) && (a<b)
5)   ris = (a+b) || (a>b)
6)   ris = (a*c-b) || (a>b)
7)   ris = ((a*c) != b) || (a>b)
8)   ris = (a>b) || (a<c) || (c==b)
```

Scrivere un programma che verifichi le risposte date.

* 6. Se le variabili intere `a`, `b` e `c` avessero gli stessi valori di partenza dell'esercizio precedente, le seguenti espressioni restituirebbero vero o falso?

```
1)   (a>b) || (c>a))
2)   (c>a) && (a>b))
3)   !(a>b) && (c>a))
4)   !(a>b) || !(c>a))
5)   (a==c) || ((a<b) && (b<c))
6)   (a!=c) || ((a<b) && (b<c))
```

Scrivere un programma che verifichi le risposte date.

* 7. Supponendo che le variabili intere `x`, `y` abbiano valori 12, 45 e che le variabili carattere `a` e `b` abbiano valori “t” e “T”, le seguenti espressioni restituirebbero vero o falso?

1) `(x>y) || (a!=b)`
2) `(y>x) && (a==b)`
3) `(a!=b) && !(x>y)`
4) `x || (y<x)`
5) `a == (b='t')`
6) `!x`

Scrivere un programma che verifichi le risposte date.

* 8. Utilizzando l’espressione condizionale `?:` scrivere un programma che, dati tre valori interi memorizzati nelle variabili `a`, `b` e `c`, assegna a `d`:

- il volume del parallelepipedo di lati `a`, `b` e `c` se il valore di `a` al quadrato sommato a `b` è diverso da `c`;
- la somma di `a`, `b` e `c`, altrimenti.

* 9. Scrivere un programma che visualizzi il seguente menu:

```
          MENU DI PROVA

        a) Per immettere dati
        b) Per determinare il maggiore
        c) Per determinare il minore
        d) Per ordinare
        e) Per visualizzare

                          Scelta: _
```

quindi attenda l’immissione di un carattere da parte dell’utente e visualizzi una scritta corrispondente alla scelta effettuata, del tipo: “`In esecuzione l'opzione a`”. Se la scelta non è tra quelle proposte (a, b, c, d, e) deve essere visualizzata la scritta: “`Opzione inesistente`”. Si utilizzi il costrutto `switch-case` e la funzione `getchar`.

* 10. Ripetere l’Esercizio 1 ma utilizzando l’espressione condizionale con l’operatore “`?:`”.

11. Scrivere un programma che, richiesto il numero `MM` rappresentante il valore numerico di un mese, visualizzi, se $1 \leq$ `MM` ≤ 12, il nome del mese per esteso, altrimenti la frase “`Valore numerico non valido`”.

12. Scrivere un programma che, richiesto il numero `AA` rappresentante un anno, verifichi se questo è bisestile. [*Suggerimento*: un anno è bisestile se è divisibile per 4 ma non per 100 (cioè si escludono gli anni-secolo).]

13. Scrivere un programma che, richiesti i numeri `GG`, `MM`, `AA` di una data, verifichi se questa è valida.

14. Scrivere il programma che, richiesti sei numeri che rappresentano due date nel formato `GG`, `MM`, `AA`, determini la più recente.

15. Scrivere un programma che, richiesti in input tre numeri interi, a seconda dei casi visualizzi una delle seguenti risposte:

```
Tutti uguali
Due uguali e uno diverso
Tutti diversi
```

3.1 Istruzione `for`

Quando si desidera ripetere una operazione un determinato numero di volte, si può riscrivere sequenzialmente l’istruzione corrispondente. Per esempio, se si vuole sommare tre volte alla variabile `somma`, inizializzata a 0, il valore 7, si può scrivere:

```
somma = 0;
```

```
     somma = somma+7;
     somma = somma+7;
     somma = somma+7;
```

Risulta però decisamente più comodo inserire in un ciclo l'istruzione che si ripete:

```
somma = 0;
     for(i=1; i<=3; i=i+1)
       somma = somma+7;
```

L'esempio precedente è volutamente semplice per concentrare l'attenzione sulle caratteristiche del costrutto `for`. Alla variabile `somma` viene dato il valore `0`. L'istruzione `for` assegna il valore `1` alla variabile `i`. L'operazione

```
i=1
```

compresa tra la parentesi tonda aperta e il primo punto e virgola è detta *inizializzazione* e non verrà mai più eseguita. Successivamente l'esecuzione prosegue così:

1. 1. se `i<=3` allora vai al passo 2 altrimenti termina
2. 2. `somma=somma+7`
3. 3. `i=i+1`, vai al passo 1

Inizialmente la condizione risulta vera, in quanto 1 è minore o uguale a 3, e quindi viene eseguito il corpo del ciclo, che in questo caso è composto dalla sola istruzione `somma=somma+7`. La variabile `somma` assume il valore `7`. Viene incrementato di 1 il valore di `i`, che quindi assume il valore `2`.
Alla fine del terzo ciclo la variabile `somma` ha il valore `21` e `i` vale `4`. Nuovamente viene verificato se `i` è minore o uguale a 3. La condizione risulta falsa e l'iterazione ha termine; il controllo passa all'istruzione successiva del programma.

Il formato del costrutto `for` è il seguente:

```
for(esp1; esp2; esp3)
  istruzione
```

Si faccia attenzione ai punti e virgola all'interno delle parentesi. Il ciclo inizia con l'esecuzione di *esp1*, la quale non verrà mai più eseguita. Quindi viene esaminata *esp2*. Se *esp2* risulta vera, viene eseguita *istruzione*, altrimenti il ciclo non viene percorso neppure una volta.

Successivamente viene eseguita *esp3* e di nuovo valutata *esp2* che se risulta essere vera dà luogo a una nuova esecuzione di `istruzione`. Il processo si ripete finché *esp3* non risulta essere falsa. Nell'esempio precedente,

```
for(i=1; i<=3; i=i+1)
  somma=somma+7;
```

esp1 era `i=1`, *esp2* `i<=3`, *esp3* `i=i+1` e *istruzione* era `somma=somma+7`.

Nella sintassi del `for`, *istruzione*, così come nel costrutto `if`, può essere un blocco, nel qual caso deve iniziare con una parentesi graffa aperta e terminare con parentesi graffa chiusa. Supponiamo di voler ottenere la somma di tre numeri interi immessi dall'utente: si può scrivere:

```
somma = 0;
scanf("%d", &numero);
somma = somma+numero;
scanf("%d", &numero);
somma = somma+numero;
scanf("%d", &numero);
somma = somma+numero;
```

La variabile `somma` che conterrà, di volta in volta, la somma degli interi letti, viene inizializzata a zero. Successivamente, per tre volte è richiesta l'immissione di un valore che viene immagazzinato nella variabile `numero`. A ogni lettura corrisponde un incremento di `somma` del valore di `numero`. Lo stesso risultato lo si ottiene in una forma più sintetica con il seguente codice:

```
somma = 0;
for(i=1; i<=3; i =i+1) {
  scanf("%d", &numero);
```

```
  somma = somma+numero;
}
```

Nel Listato 3.1 osserviamo un programma che calcola la somma di cinque numeri interi immessi dall'utente.

```
/* Esempio di utilizzo dell'istruzione for
   Calcola la somma di cinque numeri interi
   immessi dall'utente                        */

#include <stdio.h>

int i, somma, numero;

main()
{
printf("SOMMA 5 NUMERI\n");
somma = 0;

for(i=1; i<=5; i=i+1) {
  printf("Inser. intero: ");
  scanf("%d", &numero);
  somma = somma + numero;
}

printf("Somma: %d\n",somma);
}
```

Listato 3.1 Iterazione con l'istruzione `for`

Durante l'esecuzione del programma l'utente sarà chiamato a introdurre cinque valori interi:

```
SOMMA 5 NUMERI
Inser. intero: 32
Inser. intero: 111
Inser. intero: 2
Inser. intero: 77
Inser. intero: 13
Somma: 235
```

In questo caso l'utente ha inserito 32, 111, 2, 77 e 13. Naturalmente il numero dei valori richiesti può variare: se si vogliono accettare 100 valori si deve modificare soltanto la condizione di fine ciclo *esp2* nel `for`:

```
for(i=1; i<=100; i=i+1)
```

Potrebbe risultare utile far apparire il numero d'ordine d'inserimento; a tale scopo si deve modificare la prima `printf`:

```
printf("\nInser. intero n.%d: ", i);
```

che genererà

```
Inser. intero n.1: 32
Inser. intero n.2: 111
Inser. intero n.3: 2
...
```

Nel costrutto `for`

```
for(esp1; esp2; esp3)
  istruzione
```

esp1, come *esp2* ed *esp3*, può essere una qualsiasi espressione ammessa in C ▬.

Per ora limitiamoci a vederne alcune applicazioni classiche:

```
for(i=5; i>=1; i=i-1)
```

Il ciclo viene ripetuto cinque volte ma la variabile che controlla il ciclo viene inizializzata al valore massimo (5) e decrementata di uno a ogni passaggio; l'ultima iterazione avviene quando il valore assunto è `1`. Se si desidera far assumere alla variabile che controlla un ciclo, ripetuto quattro volte, i valori 15, 25, 35 e 45 si potrà scrivere

```
for(i=15 ; i<=45; i=i+10)
```

Analogamente, se i valori devono essere 7, 4, 1, –2, –5, –8 si avrà:

```
for(i=7; i>=-8; i=i-3)
```

Quando si predispone la ripetizione ciclica di istruzioni si deve fare molta attenzione a che l'iterazione non sia infinita, come nell'esempio seguente:

```
for(i=5; i>=5; i=i+1)
```

Il valore di `i` viene inizializzato a `5`; è dunque verificata la condizione `i>=5`. Successivamente `i` viene incrementato di una unità e assume di volta in volta i valori `6`, `7`, `8` ecc. che risulteranno essere sempre maggiori di `5`: il ciclo è infinito. Il compilatore non segnalerà nessun errore ma l'esecuzione del programma probabilmente non farà ciò che si desidera.

✓ **NOTA**

Situazioni di questo genere si presentano di frequente perché non è sempre banale riconoscere un'iterazione infinita; perciò si utilizzino pure le libertà linguistiche del C, ma si abbia cura di mantenere sempre uno stile di programmazione strutturato e lineare, in modo da accorgersi rapidamente degli eventuali errori commessi.

Ognuna delle *esp1*, *esp2* ed *esp3* può essere l'espressione nulla, nel qual caso comunque si deve riportare il punto e virgola corrispondente. Vedremo nei prossimi paragrafi alcuni esempi significativi. Anche l'*istruzione* del `for` può essere nulla, corrispondere cioè a un punto e virgola, come nell'esempio:

```
for(i=1; i<1000; i=i+100)
   ;
```

che incrementa il valore `i` di 100 finché `i` risulta minore di 1000. Si osservi che al termine dell'esecuzione dell'istruzione `i` avrà valore 1001: è chiaro perché?

3.2 Incrementi e decrementi

L'incremento unitario del valore di una variabile è una delle operazioni più frequenti. Si ottiene con l'istruzione

```
somma = somma+1;
```

In C è possibile ottenere lo stesso effetto mediante l'operatore `++`, costituito da due segni di addizione non separati da nessuno spazio. L'istruzione

```
++somma;
```

incrementa di uno il valore di `somma`, esattamente come faceva l'istruzione precedente. Lo stesso ragionamento vale per l'operazione di decremento, per cui

```
somma = somma-1;
```

è equivalente a

```
--somma;
```

L'operatore `--`, costituito da due segni di sottrazione, decrementa la variabile di una unità. Dunque anche l'istruzione `for`

```
for(i=1; i<=10; i=i+1)
```

può essere trasformata in

```
for(i=1; i<=10; ++i)
```

Osserviamo come viene modificato il ciclo del programma esaminato precedentemente grazie all'utilizzo dell'operatore `++`:

```
for(i=1; i<=5; ++i) {
  printf("\nInser. intero: ");
  scanf("%d", &numero);
  somma = somma+numero;
}
```

Si noti come il codice C si faccia via via più compatto.

Gli operatori `++` e `--` possono precedere una variabile in un'espressione:

```
int a, b, c;
a = 5;
b = 7;
c = ++a + b;
printf("%d \n", a);
printf("%d \n", b);
printf("%d \n", c);
```

Nell'espressione `++a+b`, la variabile `a` viene incrementata di una unità (`++a`) e sommata alla variabile `b`. Successivamente il risultato viene assegnato a `c`. Le tre istruzioni `printf` visualizzeranno rispettivamente 6, 7 e 13.

Gli operatori `++` e `--` hanno priorità maggiore degli operatori binari aritmetici, relazionali e logici, per cui vengono considerati prima degli altri (Figura 3.1).

```
!   -   ++   --
*   /   %
+   -
>   >=   <   <=
==   !=
&&
||
?:
=   +=   -=   *=   /=   %=
```

Figura 3.1 Tavola di priorità degli operatori esaminati

Gli operatori di incremento e decremento possono sia precedere sia seguire una variabile:

```
++somma;
somma++;
```

Le due istruzioni precedenti hanno lo stesso effetto, ma se gli operatori vengono utilizzati all'interno di espressioni che coinvolgono più elementi valgono le seguenti regole:

- • se l'operatore ++ (--) precede la variabile, prima il valore della variabile viene incrementato (decrementato) e poi viene valutata l'intera espressione;
- • se l'operatore ++ (--) segue la variabile, prima viene valutata l'intera espressione e poi il valore della variabile viene incrementato (decrementato).

Per esempio:

```
int a, b, c;
a = 5;
b = 7;
c = a++ + b;
printf("%d \n", a);
printf("%d \n", b);
printf("%d \n", c);
```

non produce la stessa visualizzazione della sequenza precedente. La variabile `a` viene sommata a `b` e il risultato viene assegnato a `c`, successivamente `a` viene incrementata di una unità. Le istruzioni `printf` visualizzeranno rispettivamente 6, 7 e 12. Si osservi l'identità dei due cicli `for`:

```
for(i=1; i<=3; ++i)          for(i=1; i<=3; i++)
```

poiché `esp3` è da considerarsi un istruzione a sé stante. Viceversa

```
for(i=1; ++i<=3;)            for(i=1; i++<=3;)
```

sono diversi in quanto nel caso di sinistra `i` viene incrementata prima della valutazione di *`espr2`*, per cui nel primo ciclo `i` acquista valore 2, nel secondo 3 e il terzo ciclo non verrà mai eseguito dato che `i` ha già valore `4`. Nel caso di destra `i` assume valore `4` solamente dopo il confronto operato nel terzo ciclo, che quindi verrà portato a termine; per verificarlo si provino le successive due sequenze.

```
j=0;
     for(i=1; ++i<=3;)
       printf("Ciclo: %d\n", ++j);
     printf("Cicli:%d i:%d\n", j, i);

j=0;
     for(i=1; i++<=3;)
       printf("Ciclo: %d\n", ++j);
     printf("Cicli:%d i:%d\n", j, i);
```

Le visualizzazioni prodotte saranno rispettivamente

```
Ciclo:1            Ciclo:1
     Ciclo:2            Ciclo:2
     Cicli:2 i:4        Ciclo:3
                        Cicli:3 i:5
```

È chiaro perché `i` ha valore `4` nel caso di sinistra e `5` in quello di destra?

Non è generalmente permesso in C, ed è comunque sconsigliato per gli effetti che ne possono derivare, utilizzare in un'espressione la stessa variabile contemporaneamente incrementata e non incrementata come in `a=b+(++b)`.

Nel caso che una variabile debba essere incrementata o decrementata di un valore diverso da uno, oltre che con il metodo classico

```
somma = somma+9;
```

si può usufruire dell'operatore +=:

```
somma += 9;
```

che nell'esempio incrementa di nove unità il valore di somma. La forma generalizzata è

`variabile [operatore]= espressione`

Dove `[operatore]` può essere `+ - * / %` ed *espressione* una qualsiasi espressione lecita. La forma compatta appena vista è utilizzabile quando una variabile appare sia a sinistra sia a destra di un operatore di assegnamento ed è equivalente a quella classica:

```
variabile = variabile[operatore]espressione
```

Si hanno pertanto le seguenti equivalenze.

```
Forma compatta              Forma Classica
        a *= 5;                     a = a*5;
        a -= b;                     a = a-b;
        a *= 4+b;                   a = a*(4+b);
```

L'ultima linea evidenzia quale sia la sequenza di esecuzione nella forma compatta:

1. 1. viene calcolata l'intera espressione posta a destra dell'assegnamento: `4+b`;
2. 2. viene moltiplicato il valore ottenuto per il valore della variabile posta a sinistra dell'assegnamento: `a*(4+b);`
3. 3. viene assegnato il risultato ottenuto alla variabile posta a sinistra dell'assegnamento: `a=a*(4+b)`.

Questo funzionamento è coerente con la bassa priorità degli operatori `+=`, `-=`, `*=`, `/=` e `%=` che hanno lo stesso livello dell'assegnamento semplice `=` (Figura 3.1). Per esempio, dopo la sequenza di istruzioni

```
a = 3;
b = 11;
c = 4;
c -= a*2+b;
```

La variabile `c` ha valore –13.

3.3 Calcolo del fattoriale

Utilizziamo il costrutto `for` per il calcolo del fattoriale, indicato con $n!$, di un intero n, definito da

$n! = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \ldots\ldots \cdot 2 \cdot 1$

dove 1! e 0! sono per definizione uguali a 1. Avremo, per esempio, che

```
4! = 4·3·2·1 = 24
     6! = 6·5·4·3·2·1 = 720
```

```
/* Calcolo di n! (n fattoriale) */

#include <stdio.h>

main()
{
int n, fat, m;

printf("CALCOLO DI N!\n\n");
printf("Inser. n: ");
scanf("%d", &n);

fat = n;
for(m=n; m>2; m--)
  fat = fat*(m-1);

printf("Il fattoriale di: %d ha valore: %d\n", n, fat);
}
```

Listato 3.2 Calcolo del fattoriale di *n*

Nell'ipotesi di non considerare il caso n = 0, un algoritmo possibile è quello del Listato 3.2. Se viene passato in ingresso il valore 4, `fat` assume tale valore:

```
fat = n;
```

Il ciclo `for` inizializza 4 a `m` e controlla che sia maggiore di 2. Viene eseguito una prima volta il ciclo

```
fat = fat*(m-1);
```

e `fat` assume il valore `12`. Di nuovo il controllo dell'esecuzione passa al `for` che decrementa il valore di `m` e verifica se `m>2`, cioè se `3>2`. Viene eseguito il corpo del ciclo

```
fat = fat*(m-1);
```

e `fat` assume il valore `24`. Il `for` decrementa `m` e verifica se `m>2`, cioè se `2>2`. Questa volta l'esito è negativo e le iterazioni hanno termine. Utilizzando l'operatore `*=`, al posto di `fat=fat*(m-1)` avremmo potuto scrivere

```
fat *= m-1;
```

Per considerare anche il caso in cui sia compreso il fattoriale di zero, prima di iniziare il ciclo ci si deve chiedere se `n` ha tale valore, nel qual caso il risultato è 1.

```
fat = n;
if(n==0)
  fat = 1;
else
  for(m=n; m>2; m--)
    fat = fat*(m-1);
```

L'uso della variabile `m` è necessario perché si desidera mantenere il valore iniziale di `n` per stamparlo nella `printf` finale, altrimenti se ne potrebbe fare a meno utilizzando al suo posto direttamente `n`:

```
fat = n;
if(n==0)
  fat=1;
else
```

```
for(n=n; n>2; n--)
  fat = fat*(n-1);
```

L'inizializzazione all'interno del `for n=n` è del tutto superflua, per cui si può scrivere

```
for(; n>2; n--)
  fat = fat*(n-1);
```

Questa sintassi è permessa e indica che *`esp1`* è vuota; il punto e virgola è obbligatorio. Un altro metodo è quello di eseguire le moltiplicazioni successive a partire dal basso: $n! = 2 \cdot 3 \cdot ...(n-1) \cdot n$, inizializzando `fat` a 1 e utilizzando una variabile ausiliaria intera (Listato 3.3). Si noti come con questa soluzione sia già incluso il caso di 0!. Anche questa volta invece di `fat=fat*aux` avremmo potuto scrivere `fat*=aux` ▬.

```
/* Calcolo n! (n fattoriale) */

#include <stdio.h>

main()
{
int n, fat, aux;

printf("CALCOLO DI N!\n\n");
printf("Inser. n: ");
scanf("%d", &n);

fat = 1;
for(aux=2; aux<=n; aux++) fat = fat*aux;

printf("Il fattoriale di: %d ha valore: %d\n", n, fat);
}
```

Listato 3.3 Un'altra possibilità per il calcolo del fattoriale

3.4 Istruzione `while`

Anche l'istruzione `while`, come l'istruzione `for`, permette di ottenere la ripetizione ciclica di una istruzione:

```
while(esp)
  istruzione
```

Viene verificato che *`esp`* sia vera, nel qual caso viene eseguita *`istruzione`*. Il ciclo si ripete fintantoché `esp` risulta essere vera. Naturalmente, ancora una volta, `istruzione` può essere un blocco. Riprendiamo il programma che calcola la somma dei valori immessi dall'utente e modifichiamolo in modo da controllare il ciclo con `while`:

```
i = 1;
     while(i<=5) {                    for(i=1; i<=5; i++) {
       printf("Inser. intero: ");       printf("Inser. intero: ");
       scanf("%d", &numero);            scanf("%d", &numero);
       somma = somma+numero;            somma = somma+numero;
       i++;                           }
     }
```

L'inizializzazione della variabile che controlla il ciclo deve precedere l'inizio del `while` e l'incremento della stessa variabile deve essere inserito come ultima istruzione del blocco. In generale, quando il numero d'iterazioni è noto a priori, per passare da un `for` a un `while` vale la seguente equivalenza:

```
esp1;
     while(esp2)                       for(esp1; esp2; esp3)
       corpo_del_ciclo                   corpo_del_ciclo
```

```
    esp3;
```

Nel programma precedente si poteva inserire l'incremento della variabile di controllo del ciclo all'interno della condizione logica presente tra parentesi tonde. Si ha infatti la seguente corrispondenza:

`i = 1;`
`    while(i<=5){`
`      printf("Inser. intero: ");`
`      scanf("%d", &numero);`
`      somma = somma+numero;`
`      i++;`
`    }`
	`i = 1;`
	`    while(i++<=5) {`
	`      printf("Inser. intero: ");`
	`      scanf("%d", &numero);`
	`      somma = somma+numero;`
	`    }`

Grazie all'operatore `++` la variabile `i` viene incrementata automaticamente ad ogni ciclo. È obbligatorio posporre l'operatore alla variabile perché si desidera che l'incremento venga fatto dopo il confronto tra il valore di `i` e 10. In caso contrario il numero di iterazioni sarebbe uguale a nove. Quando si deve ripetere `n` volte un ciclo la migliore soluzione è ancora un'altra:

```
i = n;
while(i--)
  corpo_del_ciclo
```

Come abbiamo visto nel capitolo precedente, la condizione logica diviene falsa quando `i` assume valore zero. Nell'esempio precedente si ha:

```
i = 5;
while(i--) {
  printf("Inser. intero: ");
  scanf("%d", &numero);
  somma = somma+numero;
}
```

Osserviamo, ancora una volta, come il codice si faccia sempre più compatto.

Trasformiamo adesso il programma in modo che la lunghezza della serie dei numeri in ingresso non sia determinata a priori ma termini quando viene inserito il valore zero. Non è possibile evidentemente risolvere il problema con una ripetizione sequenziale d'istruzioni in quanto il numero di valori non è noto, ma viene deciso a tempo d'esecuzione (Listato 3.4).

```
/* Calcola la somma dei valori interi passati dall'utente
   termina quando viene immesso il valore 0 (zero)       */

#include <stdio.h>

main()
{
int somma, numero;

printf("SOMMA NUMERI\n");
printf("zero per terminare\n");
numero = 1;
somma  = 0;
while(numero!=0) {
  printf("Inser. intero: ");
  scanf("%d", &numero);
  somma = somma+numero;
}
printf("Somma: %d\n",somma);
}
```

Listato 3.4 Esempio di utilizzo dell'istruzione `while`

Alla variabile `numero` si è assegnato il valore 1 per far in modo che il ciclo venga eseguito almeno una volta; ovviamente qualsiasi valore diverso da zero va bene. Una possibile esecuzione è la seguente:

```
SOMMA NUMERI
zero per terminare
Inser. intero: 105
Inser. intero: 1
Inser. intero: 70
Inser. intero: 0
Somma: 176
```

dove i valori passati dall'utente sono 105, 1, 70 e 0 per terminare l'inserzione.

Ogni istruzione `for` può essere sostituita da un'istruzione `while` se si ha cura di aggiungere le opportune inizializzazioni prima del ciclo e gli opportuni incrementi all'interno dello stesso. In C è vero anche l'inverso. Ogni istruzione `while` ha un suo corrispondente `for`, anche quando il numero d'iterazione non è noto a priori. Per esempio, la parte centrale del programma precedente può essere realizzata con un ciclo `for`:

```
numero = 1;                        numero = 1;
     somma = 0;                         somma = 0;
     while(numero!=0) {                 for(; numero!=0;) {
       printf("Inser. intero: ");         printf("Inser. intero: ");
       scanf("%d", &numero);              scanf("%d", &numero);
       somma = somma+numero;              somma = somma+numero;
     }                                  }
```

Infatti, come si è già evidenziato, nel costrutto `for`

```
for(esp1; esp2; esp3)
```

è possibile sostituire *esp1*, *esp2* ed *esp3* con qualsiasi espressione, nella fattispecie *esp2* corrisponde al controllo `n!=0` (`n` diverso da 0) mentre *esp1* ed *esp3* corrispondono a espressioni vuote. La presenza dei punti e virgola è naturalmente obbligatoria.

✓ **NOTA**

L'istruzione `for`, con la sua chiarezza d'intenti, l'enorme potenza e compattezza, è largamente utilizzata dai programmatori C.

Supponiamo che oltre alla somma si desideri determinare il valore massimo della sequenza in ingresso, con la limitazione che i valori debbano essere tutti positivi. Una volta inizializzata la variabile intera `max` a zero il ciclo diventa il seguente:

```
while(numero!=0) {
  printf("Inser. intero positivo: ");
  scanf("%d", &numero);
  if(numero>max)  max=numero;
    somma = somma+numero;
}
```

All'interno di un blocco è lecito inserire qualsiasi istruzione, quindi anche un `if`. La variabile `max` viene inizializzata a zero, che è minore di qualsiasi valore che l'utente possa inserire. A ogni iterazione del ciclo viene controllato se il valore inserito dall'utente, presente nella variabile `numero`, è maggiore di `max`, nel qual caso viene assegnato a `max` il nuovo valore. Se si desidera che i valori passati in ingresso non siano comunque superiori a certo numero, supponiamo 10, si può inserire una variabile contatore degli inserimenti e controllarne il valore all'interno del `while`:

```
while(numero!=0 && i<=10)
```

Le due condizione logiche sono poste in AND, affinché l'iterazione continui: deve essere vero che `numero` è diverso da zero e che `i` è minore di 10 (Listato 3.5).

```
/* Determina somma e maggiore dei valori immessi */

#include <stdio.h>

main()
{
int somma,numero,max,i;

printf("SOMMA E MAGGIORE\n");
printf("zero per finire\n");
numero = 1;
somma = 0;
max = 0;

i = 1;
while(numero!=0 && i<=10)
{
  printf("Valore int.: ");
  scanf("%d", &numero);
  if(numero>max)
    max = numero;
  somma = somma+numero;
  i++;
}
printf("Somma: %d\n", somma);
printf("Maggiore: %d\n", max);
}
```

Listato 3.5 Diramazione `if` all'interno di una iterazione `while`

L'incremento della variabile che conta il numero di valori immessi può essere inserito direttamente nella parte *`espressione`* di `while`:

```
while(numero!=0 && i++<=10) {
  printf("Inser. intero positivo: ");
  scanf("%d", &numero);
  if(numero>max)  max=numero;
  somma+=numero;
}
```

L'incremento deve avvenire dopo il controllo `i<10`, per cui l'operatore `++` deve seguire e non precedere `i`. Il ciclo `while` esaminato nell'ultimo programma può essere, come sempre, realizzato con un `for`

```
for(i=1; numero!=0 && i<=10; i++) {
  printf("Inser. intero positivo: ");
  scanf("%d", &numero);
  if(numero>max)  max=numero;
  somma+=numero;
}
```

Per far in modo che il programma comprenda anche il caso di numeri negativi, si deve provvedere all'immissione del primo dato in `max` anteriormente all'inizio del ciclo. Una soluzione alternativa è di inizializzare `max` al minimo valore negativo accettato da una variabile intera. Nell'ipotesi che fossero riservati quattro byte a un `int` potremmo quindi scrivere:

```
max = - 2147483648;
```

ma questo valore è dipendente dall'implementazione. Nella libreria `limits.h` sono definiti i valori limite definiti dall'implementazione; in essa sono presenti alcune costanti, fra cui `INT_MAX`, che contiene il massimo valore di un `int`, e `INT_MIN`, che contiene il minimo valore di un `int`. È sufficiente includere nel programma tale libreria per poter utilizzare le variabili in essa definite:

```
#include <limits.h>
```

Si potrà inizializzare `max` al minor intero rappresentabile con una variabile di tipo `int`:

max = INT_MIN;

3.5 Istruzione do-while

Quando l'istruzione compresa nel ciclo deve essere comunque eseguita almeno una volta, risulta più comodo utilizzare il costrutto

```
do
  istruzione
while(esp);
```

Viene eseguita *istruzione* e successivamente controllato se *esp* risulta essere vera, nel qual caso il ciclo viene ripetuto. Come sempre, l'iterazione può coinvolgere una istruzione composta (blocco).

Riprendiamo il programma che determina la somma e il maggiore tra i numeri immessi dall'utente e realizziamo il ciclo centrale con l'istruzione appena vista (Listato 3.6).

```
/* Determina somma e maggiore dei valori immessi
   (esempio uso do-while)                          */

#include <stdio.h>

main()
{
int somma,numero,max,i;
```

```
printf("SOMMA E MAGGIORE\n");
printf("zero per finire\n");
numero = 1;
somma = 0;
max = 0;

i = 1;
do {
  printf("Valore int.: ");
  scanf("%d", &numero);
  if(numero>max)
    max = numero;
  somma = somma+numero;
  i++;
}
while(numero!=0 && i<=10);

printf("Somma: %d\n", somma);
printf("Maggiore: %d\n", max);
}
```

Listato 3.6 Esempio di utilizzo del costrutto `do-while`

Il ciclo viene ripetuto fine a quando la condizione del `while` risulta essere vera, o in altre parole si esce dal ciclo quando la condizione del `while` risulta essere falsa. Per trasformare un `while` in `do-while` si deve semplicemente porre il `do` all'inizio del ciclo e il `while(`*esp*`)` alla fine dello stesso. Il ciclo poteva essere più sinteticamente espresso come segue:

```
do {
  printf("Valore int.: ");
  scanf("%d", &numero);
  if(numero>max)
    max = numero;
  somma = somma+numero;
}
while(numero!=0 && ++i<=10);
```

In cui l'operatore ++ deve obbligatoriamente precedere il nome della variabile in quanto l'incremento deve avvenire prima del controllo `i<=10`.

3.6 L'operatore virgola

L'operatore virgola, che ha priorità più bassa di tutti gli altri, permette di inserire all'interno delle espressioni più istruzioni. Per esempio, un `for` può includere le inizializzazioni all'interno di *esp1*:

```
for(numero=1, somma=0; numero!=0;)
```

In questo caso *esp3* non è presente, ma se necessario anch'essa potrebbe contenere più di un istruzione:

```
for(i=1, j=5; i<10 && j<100; i++, j=i*j)
```

Nell'esempio, `i` viene inizializzato a 1 e `j` a 5. Il ciclo si ripete finché `i` è minore di 10 e contemporaneamente `j` è minore di 100. A ogni ciclo `i` viene incrementato di 1 e a `j` viene assegnato il prodotto di `i` per `j`. Al limite si potrebbe scrivere:

```
for(numero=1, somma=0; numero!=0; printf("Inser. intero:\t),
scanf("%d",&numero), somma=somma +numero)
  ;
```

comprendendo tutte le istruzioni che costituiscono il calcolo della somma dei numeri introdotti dall'utente all'interno di *esp3*. Vale la pena sottolineare che le istruzioni in *esp3* sono inframmezzate dalla virgola e non devono essere presenti punti e virgola.

✓ **NOTA**
Questo modo di operare porta a istruzioni lunghissime, difficilmente leggibili; consigliamo pertanto di usare l'operatore virgola essenzialmente là dove ci siano da inizializzare o incrementare più variabili che controllano il ciclo.

3.7 Cicli annidati

In un blocco `for` o `while`, così come nei blocchi `if-else`, può essere presente un numero qualsiasi di istruzioni di ogni tipo. Si sono visti esempi di cicli all'interno di costrutti `if` e viceversa, ora vediamo un esempio di cicli innestati uno nell'altro.

Per ripetere una determinata istruzione *n*m* volte possiamo scrivere

```
for(i=1; i<=n; i++)
  for(j=1; j<=m; j++)
     printf("i: %d j: %d \n", i, j);
```

Se prima dell'inizio del ciclo `n` ha valore 2 e `m` ha valore 3 l'esecuzione provocherà la seguente visualizzazione:

```
i: 1    j: 1
i: 1    j: 2
i: 1    j: 3
i: 2    j: 1
i: 2    j: 2
i: 2    j: 3
```

Alla variabile `i` viene assegnato il valore 1 e si esegue il ciclo interno in cui la variabile `j` assume via via i valori 1, 2 e 3; a questo punto l'istruzione di controllo del ciclo interno appura che `j` non sia minore o uguale a `m` (4<=3) e il controllo ripassa al ciclo esterno; `i` assume il valore 2 e si ripete l'esecuzione del ciclo interno.

Se desideriamo produrre sul video una serie di `n` linee e `m` colonne costituite dal carattere +, possiamo scrivere il programma del Listato 3.7.

```
#include <stdio.h>

main()        /* esempio cicli annidati */
{
int n, m, i, j;

printf("Inserire il numero di linee: \t");
scanf("%d", &n);
printf("Inserire il numero di colonne: \t");
scanf("%d", &m);
```

```
for(i=1; i<=n; i++) { /* inizio blocco ciclo esterno */
  printf("\n");
  for(j=1; j<=m; j++)
    printf("+");
}                      /* fine blocco ciclo esterno */
}
```

Listato 3.7 Esempio di cicli annidati

L'istruzione `printf("\n")` viene eseguita all'interno del ciclo grande e permette di saltare a linea nuova.

3.8 Salti condizionati e incondizionati

L'istruzione `break` consente di interrompere l'esecuzione del `case`, provocando un salto del flusso di esecuzione alla prima istruzione successiva. Una seconda possibilità di uso di `break` è quella di forzare la terminazione di un'iterazione `for`, `while` o `do-while`, provocando un salto alla prima istruzione successiva al ciclo.

Riprendiamo il programma per il calcolo della somma e la ricerca del massimo dei numeri immessi dall'utente. L'istruzione di controllo del ciclo ne bloccava l'esecuzione se veniva immesso in ingresso il valore zero o quando fossero già stati inseriti dieci valori. Scorporiamo quest'ultima verifica affidandola a un'altra istruzione:

```
for(i=1; numero!=0; i++) {
  printf("Inser. intero positivo: \t");
  scanf("%d", &numero);
  if(numero>max)  max=numero;
  somma+=numero;
  if(i==10) break;
}
```

Naturalmente era migliore la soluzione precedente:

```
for(i=1; numero!=0 && i<=10; i++)
```

L'istruzione `break` provoca l'uscita solo dal ciclo più interno. Per esempio la sequenza di istruzioni

```
for(j=1; j<=3; j++) {
  somma = 0;
  max = 0;
  numero = 1;
  for(i=1; numero!=0; i++) {
    printf("Inser. intero positivo: \t");
    scanf("%d", &numero);
    if(numero>max)  max=numero;
    somma+=numero;
    if(i==10) break;
  }
  printf("Somma: %d\n", somma);
  printf("Maggiore: %d\n", max);
}
```

ripete per tre volte il calcolo della somma e la ricerca del massimo visti precedentemente.

L'istruzione `continue` lavora in modo simile al `break`, con la differenza che anziché forzare l'uscita dal ciclo provoca una nuova iterazione a partire dall'inizio, saltando le istruzioni che rimangono dal punto in cui si trova `continue` alla fine del ciclo, come nel seguente esempio.

```
for(i=1; numero!=0 && i<=10; i++) {
       printf("Inser. intero positivo: \t");
       scanf("%d", &numero);
```

```
        if(numero<0) continue;
        if(numero>max)  max=numero;
        somma+=numero;
      }
```

Se il numero immesso dall'utente è negativo vengono saltate le ultime due istruzioni del ciclo.

Per analogia presentiamo la funzione `exit`, che fa parte della libreria standard `stdlib.h` e che provoca l'immediata terminazione del programma e il ritorno al sistema operativo. Normalmente la funzione viene chiamata non passandole nessun argomento, il che significa *terminazione normale*. Altri argomenti consentono di indicare che si è verificato un particolare tipo di errore e il destinatario di questa comunicazione dovrebbe essere un processo di livello superiore in grado di gestire la condizione anomala. Nell'esempio precedente, in caso di immissione da parte dell'utente del valore zero, il programma sarebbe terminato:

```
if(numero<0) exit();
```

Generalmente `exit` viene inserita dopo la verifica negativa di una condizione indispensabile per il proseguimento dell'esecuzione. Per ipotesi, un programma relativo a un gioco potrebbe richiedere la presenza di una scheda grafica nel sistema per essere eseguito:

```
if(!scheda_grafica()) exit;
gioco();
```

`scheda_grafica` è una funzione definita dall'utente che ha valore vero se almeno una delle schede grafiche richieste è presente e falso in caso contrario. In quest'ultimo caso viene eseguito `exit` e il programma ha termine ▬.

L'istruzione `goto` richiede un'*etichetta* – un identificatore C valido – seguito da un carattere due punti. Tale identificatore deve essere presente nell'istruzione `goto`:

```
i=1;
ciclo:
  i++;
  printf("Inser. intero positivo: \t");
  scanf("%d", &numero);
  if(numero>max)  max=numero;
  somma+=numero;
if(numero!=0 && i<=10) goto ciclo;
```

Le ragioni della programmazione strutturata, tra cui pulizia ed eleganza del codice, sconsigliano l'uso generalizzato di `break`, `continue` ed `exit`, e "proibiscono" quello del `goto`.

3.9 Variabili di tipo virgola mobile

I numeri che hanno una parte frazionaria sono detti in virgola mobile (*floating point*). Per esempio:

```
152.23
-91.64
0.867
```

non possono essere memorizzati nelle variabili di tipo `int`. Le variabili che contengono tali valori sono di tipo `float`:

```
float x, y, z;
```

La parola chiave `float` specifica che gli identificatori `x`, `y` e `z` che la seguono si riferiscono a variabili in virgola mobile. La definizione fa sì che venga riservato uno spazio in memoria la cui dimensione può variare rispetto all'implementazione, ma che spesso è di 4 byte (32 bit), sufficiente per contenere numeri che vanno da 3.4E-38 a 3.4E+38, cioè valori positivi e negativi che in modulo sono compresi approssimativamente tra 10 elevato alla −38 e 10 alla +38. Le seguenti istruzioni assegnano valori a variabili `float`:

```
x = 152.23;
y = 0.00008;
z = 7E+20;
```

La seconda istruzione poteva essere scritta anche come `y=.00008`, dove lo zero prima del punto decimale viene sottinteso. Il valore assegnato a `z` è in notazione esponenziale e va letto come $7*10^{20}$. La lettera `E`, che può essere anche minuscola, indica che il numero che la precede deve essere moltiplicato per 10 elevato al numero che la segue. L'uso della notazione esponenziale da parte del programmatore risulta comoda quando il numero da rappresentare è o molto grande o molto piccolo; infatti avremmo anche potuto scrivere

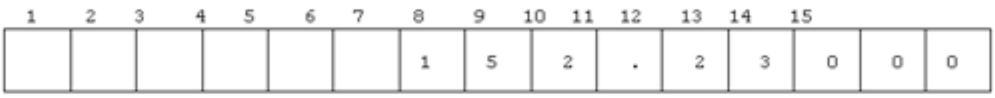
```
z = 700000000000000000000;
```

ma avremmo ottenuto certamente qualcosa di meno leggibile. Per rappresentare la costante fisica di Plank che ha valore 0.00000000000000000000000000000000006626 Js, è sufficiente scrivere 6.626E−34 Js.

Per visualizzare una variabile `float` all'interno del formato della `printf` si deve specificare dopo il solito simbolo `%` il carattere `f` se si desidera il numero decimale in virgola mobile, e il carattere `e` se lo si desidera in forma esponenziale (detta anche notazione scientifica). Come per i numeri interi, si può far seguire il simbolo di percentuale da un numero che specifica la lunghezza del campo in cui dovrà essere posto il valore opportunamente allineato a destra. Per esempio, `printf("%15f", x)`, riserva quindici caratteri per la stampa di `152.23`, che verrà posto negli ultimi sette caratteri della maschera. La lunghezza del campo può essere seguita da un punto e dal numero di caratteri del campo che devono essere riservati alla parte decimale. Per esempio:

```
printf("%15.5f", x);
```

riserva 5 caratteri per la parte decimale dei 15 totali.



Se la parte decimale non entra completamente nel sottocampo a lei riservato, le cifre meno significative vengono perdute. Al contrario, se la parte intera è più grande, il campo viene esteso fino a contenerla tutta. Se si scrive `%.0` o `%0.0` la parte decimale non viene visualizzata.

Come per gli `int`, un carattere `-`, dopo il simbolo di percentuale e prima della specifica del campo, indica che il valore deve essere allineato a sinistra.

Le istruzioni:

```
printf("%15.5f", x);    printf("%f", x);      printf("%e", x);
     printf("%15.5f", y);    printf("%f", y);      printf("%e", y);
```

restituiscono rispettivamente

```
......152.23000          152.229996           1.522300e+002
........0.00008          0.000080             8.000000e-005
```

Esiste anche una terza possibilità, data dal `%g`, che stampa la rappresentazione più breve tra `%f` e `%e`, eliminando eventualmente gli zeri superflui, per cui

```
printf("%g", x);
printf("%g", y);
```

visualizzano

```
152.23
8e-005
```

In memoria le variabili `float` vengono comunque rappresentate in una particolare notazione esponenziale, in modo da risparmiare spazio. Naturalmente i numeri reali sono infiniti mentre i sistemi di elaborazione devono fare i conti con le limitazioni fisiche proprie della macchina; dunque dobbiamo fare attenzione: stiamo lavorando con delle approssimazioni che calcoli successivi possono rendere inaccettabili. Per far in modo che la rappresentazione dei reali sia ulteriormente più precisa il C ha un altro tipo di dato in virgola mobile, detto `double`, che occupa uno spazio generalmente di 8 byte (64 bit) e che quindi permette di lavorare con numeri positivi e negativi nell'intervallo da 1.7E-308 a 1.7E+308.

Il tipo `double` comunque non garantisce automaticamente di poter rappresentare un numero doppio di cifre significative, ma certamente migliora l'accuratezza delle operazioni aritmetiche e riduce l'effetto degli errori di arrotondamento.

Per poter visualizzare una variabile `double` con la `printf` si può utilizzare nel formato la solita notazione `%f` o la sequenza `%lf` (*long float*), con le stesse convenzioni viste per le variabili `float`. L'istruzione

```
scanf("%f", &x);
```

memorizza il valore passato dall'utente nella variabile `float x`. Al posto di `%f` si può utilizzare indifferentemente `%e`. Analogamente per le variabili `double` si usa `%f` o `%lf`.

Ogni definizione di costante che includa un punto decimale fa sì che venga creata una costante di tipo `double`:

```
#define  PI 3.14159
```

definisce la costante `PI` che può essere utilizzata all'interno del programma al posto del valore `3.14159`; naturalmente il valore di `PI` non può essere modificato.

3.10 Operazioni in virgola mobile

Le operazioni aritmetiche permesse sulle variabili `float` e `double` sono le stesse che per gli `int`, e si possono scrivere espressioni con variabili di tipo misto. In ogni espressione dove compaia una variabile `float` (`double`) il calcolo viene eseguito considerando le parti frazionarie in precisione semplice (doppia). Naturalmente quando si va ad assegnare il valore ottenuto a una variabile, se essa è di precisione inferiore al risultato può succedere che ciò che si ottiene non sia significativo. Per esempio, date le seguenti dichiarazioni

```
int i;
float x;
double y;
```

se alle richieste

```
printf("\n\n Digitare un valore reale: ");
scanf("%f", &x);
printf("\n\n Digitare un valore intero: ");
scanf("%d", &i);
```

l'utente immette i valori 152.23 e 7, dopo gli assegnamenti

```
y = x;
x = i+x;
i = i+y;
```

la `printf`

```
printf("\n valore di x: %.2f  valore di i: %d", x, i);
```

visualizzerà i valori:

```
valore di x: 159.23  valore di i: 159
```

Ovviamente l'assegnamento di `i+x` a `i` fa perdere al risultato la parte decimale. Ma se venissero immessi `i` valori 56489.45 e 7, la visualizzazione sarebbe (dipende comunque dall'implementazione C):

```
valore di x: 56496.45  valore di i: -9040
```

Quindi si utilizzino operazioni miste ma coscienziosamente.

Per ottenere la massima precisione possibile le funzioni matematiche che si trovano nella libreria `math.h` solitamente acquisiscono in ingresso e restituiscono valori `double`. Tra esse le trigonometriche

```
sin(x)        seno
cos(x)        coseno
tan(x)        tangente
sinh(x)       seno iperbolico
cosh(x)       coseno iperbolico
```

e altre di uso generale come

```
log(x)        logaritmo in base e di x
log10(x)      logaritmo in base 10 di x
sqrt(x)       radice quadrata
```

L'uso di variabili di alta precisione dovrebbe comunque essere limitato ai casi di effettiva utilità in quanto utilizzare un `float` al posto di un `int` o un `double` al posto di uno degli altri due tipi, oltre a portare a una maggiore occupazione di memoria, determina una maggior lavoro di elaborazione delle operazioni e quindi diminuisce i tempi di risposta. Inoltre, a nostro avviso, usare una variabile `float` invece di un `int` dove non sia necessario porta a una peggiore leggibilità dei programmi.

3.11 Zero di una funzione

Per esercitarci con le strutture iterative e i numeri reali prendiamo in considerazione il problema del calcolo dello zero di una funzione continua $f(x)$ con il cosiddetto metodo *dicotomico*. Ricordiamo che si dice *zero* di f un numero x_0 tale che $f(x_0)=0$.

Sia $f(x)$ una funzione continua che negli estremi dell'intervallo $[a,b]$ assume valori di segno discorde, ovvero uno negativo e uno positivo e quindi tale che $f(a)*f(b)<0$, come mostrato in Figura 3.2. Sia $m=(a+b)/2$ il punto medio dell'intervallo; se $f(m)=0$ abbiamo ottenuto lo zero cercato, altrimenti si considera:

- l'intervallo $[a,m]$ se $f(a)$ e $f(m)$ hanno segno discorde;
- l'intervallo $[m,b]$ se $f(a)$ e $f(m)$ hanno segno concorde.

Si itera quindi lo stesso procedimento nell'intervallo appena determinato e si prosegue fino a trovare uno zero di f.

È importante osservare come per la corretta soluzione del problema sia indispensabile che $f(a)$ e $f(b)$ abbiano segno discorde. È quindi opportuno, prima di effettuare i calcoli relativi alla determinazione dello zero, controllare la validità della condizione nel segmento $[a,b]$ preso in esame. Non è difficile implementare quest'algoritmo usando, come esempio, la funzione

$$f(x) = 2x^3-4x+1$$

considerando che $f(0)=+1$ e $f(1)=-1$ sono di segno discorde e quindi l'intervallo $[0,1]$ soddisfa le ipotesi fatte. Il programma è quello fornito nel Listato 3.8, dove utilizziamo la funzione `fabs` che, come la funzione `abs`, calcola il valore assoluto di un numero, ma può essere applicata a valori di tipo `float`.

```
/* Determina lo zero della funzione f(x) = 2x3-4x+1 */

#include <stdio.h>
#include <math.h>

#define ERR 0.001
```

```
main()
{
float a, b, m;
float  fa, fb, fm;
char x;

/* controllo validità a, b */
do {
  printf("Inserire a: ");
  scanf("%f", &a);
  printf("Inserire b: ");
  scanf("%f", &b);
  fa = 2*a*a*a-4*a+1; /* Calcolo della funzione per x=a */
  fb = 2*b*b*b-4*b+1; /* Calcolo della funzione per x=b */
}
while(fa*fb>0);

/* calcolo zero f */
do {
  m  = (a+b)/2;
  fm = 2*m*m*m-4*m+1;   /* Calcolo della funzione per x=m */
  if(fm!=0) {
    fa = 2*a*a*a-4*a+1; /* Calcolo della funzione per x=a */
    fb = 2*b*b*b-4*b+1; /* Calcolo della funzione per x=b */
    if(fa*fm<0) b=m; else a=m;
    fm = 2*m*m*m-4*m+1;  /* Calcolo della funzione per x=m */
  }
}
while(fabs(fm) > ERR);

printf("Zero di f in %7.2f\n", m);
}
```

Listato 3.8 Programma per il calcolo dello zero di una funzione

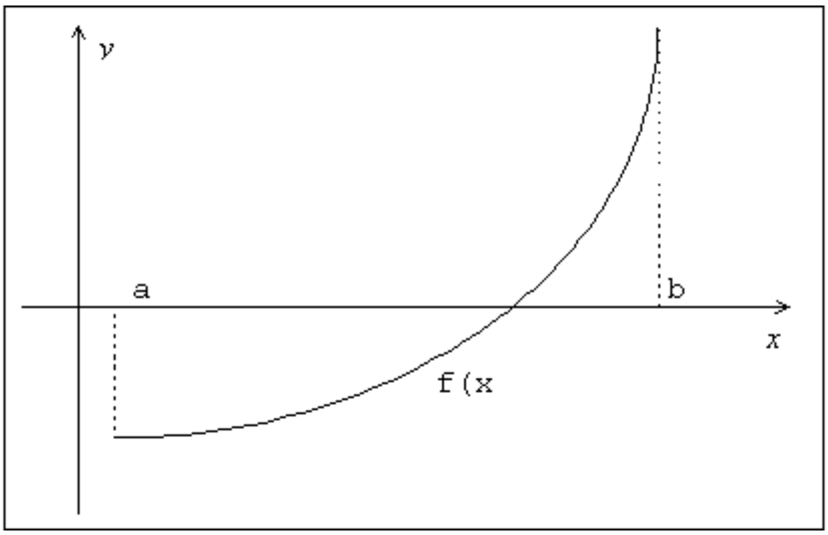


Figura 3.2 La funzione continua f assume segno discorde negli estremi dell'intervallo [a,b]

È importante osservare che a causa delle approssimazioni effettuate dalla macchina nell'esecuzione dei calcoli in genere non si ottiene uno zero effettivo della funzione ma solo una sua buona approssimazione. Per questa ragione nel `do-while` che controlla il ciclo di ricerca dello zero la condizione di controllo è `fabs(fm)>ERR` e non `fm=0`.

Effettuando il confronto con un errore ancora più piccolo (per esempio ERR=`1E-10`) si migliora il livello di approssimazione anche se questo può richiedere un tempo di calcolo molto maggiore. Abbiamo usato la funzione `fabs` che calcola il valore assoluto di un numero reale così come abbiamo visto `abs` fare di un intero.

Anticipiamo comunque che, una volta acquisite le conoscenze necessarie per creare noi stessi degli specifici sottoprogrammi, è possibile risolvere questo problema in modo molto più brillante e conciso.

Giustifichiamo infine il termine con cui è noto questo tipo di ricerca dello zero di una funzione ricordando che l'aggettivo dicotomico deriva da una parola greca che significa dividere a metà.

3.12 Esercizi

* 1. Predisporre un programma che calcola il valore dei fattoriali di tutte i numeri interi minori uguali a *n*.

* 2. Predisporre un programma, che determini il maggiore, il minore e la media degli *n* valori immessi dall'utente.

* 3. Predisporre un programma che stampi un rettangolo la cui cornice sia costituita da caratteri asterisco, e la parte interna da caratteri Q. Il numero di linee e di colonne del rettangolo viene deciso a tempo di esecuzione dall'utente; per esempio se il numero delle linee è uguale a 5 e il numero di colonne a 21, sul video deve apparire:

```
*********************
*QQQQQQQQQQQQQQQQQQQ*
*QQQQQQQQQQQQQQQQQQQ*
*QQQQQQQQQQQQQQQQQQQ*
*********************
```

* 4. Ripetere l'esercizio 3 ma permettendo all'utente di decidere anche i caratteri che devono comporre la cornice e la parte interna del rettangolo e quante volte debba essere ripetuta la visualizzazione del rettangolo.

5. Realizzare un programma che richieda all'utente *n* interi, e visualizzi il numero di volte in cui sono stati eventualmente immessi i valori 10, 100 e 1000.

6. Predisporre un programma che visualizzi la tavola pitagorica del sistema di numerazione decimale.

7. Scrivere un programma che visualizzi tutte le coppie di numeri presenti sulla superficie dei pezzi del domino.

8. Supponiamo che `x`, `y`, `z` e `t` siano variabili di tipo `float` e che `a`, `b` e `c` siano di tipo `int`. Determinare il valore di `a` e `x` dopo l'esecuzione delle seguenti istruzioni.

```
y = 2.4;
z = 7.0;
b = 3;
c = 7;
t = 0.1E2;
a = b*c + t/z;
x = a/c + t/z*y;
```

9. Tradurre i seguenti numeri in rappresentazione decimale nella corrispondente notazione esponenziale:
   a) 123.456;
   b) 2700000;
   c) 0.99999999;
   d) 0.07.

10. Modificare il programma del presente capitolo che calcola lo zero della funzione in modo che consideri

$y=x^3-2$

nell'intervallo (0,2) con un errore minore di 0,00001.

10. Utilizzare il precedente esercizio per scrivere un programma che calcoli la radice cubica di 2.

Generalizzare quindi il programma così da calcolare la radice cubica di un qualsiasi `float` immesso dall'utente.

11. Scrivere un programma che calcoli la radice *n*-esima di `a`, con *n* e `a` richiesti in input all'utente e *n*<=10. [Sugg.: modificare opportunamente il precedente esercizio.]

4.1 Array

Quando si ha la necessità di trattare un insieme omogeneo di dati esiste una soluzione diversa da quella di utilizzare tante variabili dello stesso tipo: definire un *array*, ovvero una variabile strutturata dove è possibile memorizzare più valori tutti dello stesso tipo. Intuitivamente, un array monodimensionale o *vettore* può essere immaginato come un contenitore suddiviso in tanti scomparti quanti sono i dati che vi si vogliono memorizzare. Ognuno di questi scomparti, detti *elementi* del vettore, contiene un unico dato ed è individuato da un numero progressivo, detto *indice*, che specifica la posizione dell'elemento all'interno del vettore stesso. L'indice può assumere valori interi da zero al numero totale di elementi meno 1. L'indice di base dell'array è sempre zero. Il numero complessivo degli elementi del vettore viene detto *lunghezza*.

In Figura 4.1, per esempio, l'elemento di indice 2 o, più brevemente, il terzo elemento del vettore contiene il carattere `Q`, il carattere `S` è contenuto nell'elemento di indice 0 e il vettore ha lunghezza 4.

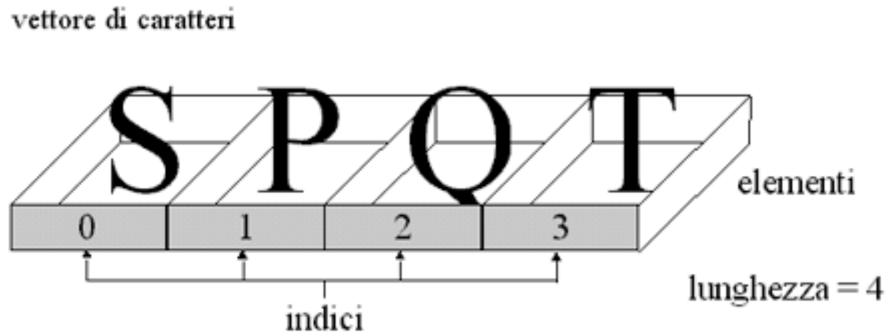


Figura 4.1 Rappresentazione intuitiva di un vettore

In definitiva, un vettore è una struttura di dati composta da un numero determinato di elementi tutti dello stesso tipo, ognuno dei quali è individuato da un indice specifico. È ora chiaro perché i vettori si dicano *variabili strutturate* mentre all'opposto tutte le variabili semplici siano anche dette *non strutturate*. Il tipo dei dati contenuti nel vettore viene detto *tipo del vettore*, ovvero si dice che il vettore è di quel particolare tipo.

Dunque per il vettore, come per qualsiasi altra variabile, devono essere definiti il nome e il tipo; inoltre si deve esplicitarne la lunghezza, cioè il numero di elementi che lo compongono. Una scrittura possibile è perciò la seguente:

```
int a[6];
```

Come sempre in C, prima deve essere dichiarato il tipo (nell'esempio `int`), poi il nome della variabile (`a`), successivamente – tra parentesi quadre – il numero degli elementi (`6`) che dev'essere un intero positivo. Questa dichiarazione permette di riservare in memoria centrale uno spazio strutturato come in Figura 4.2.

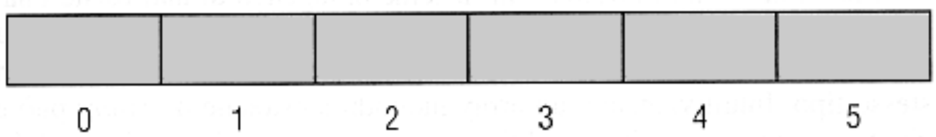


Figura 4.2 Struttura dell'array `a[6]`

Per accedere a un singolo elemento di `a` si deve specificare il nome del vettore seguito dall'indice dell'elemento posto tra parentesi quadre. L'array `a` è composto da sei elementi e l'indice può quindi assumere i valori: 0, 1, 2, 3, 4, e 5. Le istruzioni

```
a[0] = 71;
     a[1] = 4;
```

assegnano al primo elemento del vettore `a` il valore 71 e al secondo 4. Se `b` è una variabile intera (cioè dello stesso tipo del vettore), è possibile assegnare il suo valore a un elemento di `a` e viceversa:

```
a[3] = b;
```

In generale un singolo elemento dell'array può essere utilizzato esattamente come una variabile semplice. Nell'espressione

```
b = b + a[0] * a[5];
```

il valore di `b` è sommato al prodotto tra il primo e il sesto elemento di `a` e il risultato è assegnato a `b`.

Spesso l'array viene trattato all'interno di iterazioni; infatti risulta semplice far riferimento a suoi elementi incrementando ciclicamente il valore di una variabile intera e utilizzandola come indice. Consideriamo un'iterazione di esempio:

```
/* Inizializzazione dell'array */
     for(i=0; i<=5; i++) {
       printf("Inser. intero: ");
```

```
      scanf("%d", &a[i]);
    }
```

L'indice `i` dell'array `a` è inizializzato a 0 (si veda il Paragrafo 4.3) e assume a ogni iterazione successiva i valori 1, 2, 3, 4, 5. Il blocco del `for` richiede all'utente l'immissione di sei valori che vengono assegnati sequenzialmente, mediante l'istruzione `scanf`, agli elementi del vettore. Se quindi vengono inseriti in sequenza i valori 9, 18, 7, 15, 21 e 11, dopo l'esecuzione del ciclo il vettore si presenterà in memoria come in Figura 4.3.

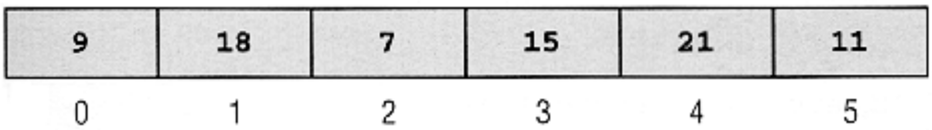


Figura 4.3 L'array `a[6]` dopo la sua inizializzazione

Anche per ricercare all'interno dell'array valori che soddisfano certe condizioni si utilizzano abitualmente i cicli:

```
/* Ricerca del maggiore */
     max = a[0];
     for(i=1; i<=5; i++)
       if(a[i]>max) max = a[i];
```

L'esempio premette di determinare il maggiore degli elementi dell'array `a`: la variabile `max` viene inizializzata al valore del primo elemento del vettore, quello con indice zero. Successivamente ogni ulteriore elemento viene confrontato con `max`: se risulta essere maggiore, il suo valore viene assegnato a `max`. Il ciclo deve comprendere dunque tutti gli elementi del vettore meno il primo, perciò l'indice `i` assume valori che vanno da 1 a 5.

Nel Listato 4.1 è riportato un programma che richiede all'utente l'immissione del punteggio raggiunto da sei studenti, li memorizza nel vettore `voti` e ne determina il maggiore, il minore e la media.

```
/* Memorizza in un array di interi il punteggio raggiunto da sei
   studenti e ne determina il maggiore, il minore e la media    */

#include <stdio.h>

main()
{
int voti[6];
int i, max, min;
float media;

printf("VOTI STUDENTI\n\n");
/* Immissione voti */
for(i=0; i<=5; i++) {
  printf("Voto %d° studente: ", i+1);
  scanf("%d", &voti[i]);
}

/* Ricerca del maggiore */
max = voti[0];
for(i=1; i<=5; i++)
  if(voti[i]>max)
    max = voti[i];

/* Ricerca del minore */
min = voti[0];
for(i=1; i<=5; i++)
  if(voti[i]<min)
    min = voti[i];
```

```
/* Calcolo della media  */
media = voti[0];
for(i=1; i<=5; i++)
  media = media + voti[i];
media = media/6;

printf("Maggiore: %d\n", max);
printf("Minore: %d\n", min);
printf("Media: %f\n", media);
}
```

Listato 4.1 Esempio di utilizzo di una variabile array

Si noti che la richiesta dei voti all'utente viene fatta evidenziando il numero d'ordine che corrisponde al valore dell'indice aumentato di una unità.

```
printf("Voto %d° studente: ", i+1);
```

Alla prima iterazione appare sullo schermo:

```
Voto 1° studente:
```

Considerazioni analoghe a quelle fatte per il calcolo del maggiore valgono per il minimo e la media.

Nella pratica la memorizzazione in un vettore ha senso quando i valori debbono essere utilizzati più volte, come nel caso precedente. Nell'esempio, comunque, i calcoli potevano essere effettuati all'interno della stessa iterazione con un notevole risparmio di tempo di esecuzione: ▬

```
/* Ricerca maggiore, minore  e media */
     max = voti[0];
     min = voti[0];
     media = voti[0];
     for(i = 0; i <= 5; i++) {
       if(voti[i] > max)
         max = voti[i];
       if(voti[i] < min)
         min = voti[i];
       media = media+voti[i];
     }
     media = media / 6;
```

✓ **NOTA**

È importante ricordare che in C l'indice inferiore del vettore è zero e quello superiore è uguale al numero di elementi meno 1: se si desidera un array di 100 si dichiara

```
voti[100];
```

ma si deve tenere presente che l'indice assume valori da 0 a 99.
Uno degli errori più ricorrenti nella programmazione in C è lo sconfinamento dei limiti degli array. Non sempre è semplice rintracciare tali errori poiché i compilatori non li segnalano. I controlli necessari in questo senso sono a carico del programmatore.

Dopo la dichiarazione del tipo possono essere presenti i nomi di più variabili di quel tipo, semplici o strutturate. Per esempio

```
int i, voti[6], max, min, somma;
```
dichiara le variabili intere `i`, `max`, `min`, `somma` e la variabile array di tipo intero `voti` ▬.

4.2 Esempi di uso di array

Per determinare la destrezza di *n* concorrenti sono state predisposte due prove, entrambe con una valutazione che varia da 1 a 10; il punteggio totale di ogni concorrente è dato dalla media aritmetica dei risultati delle due prove. Si richiede la visualizzazione di una tabella che contenga su ogni linea i risultati parziali e il punteggio totale di un concorrente. Nel Listato 4.2 è mostrato il programma relativo.

```
/* Carica i punteggi di n concorrenti su due prove
   Determina la classifica                          */

#include <stdio.h>

#define MAX_CONC 1000    /* massimo numero di concorrenti */
#define MIN_PUN  1       /* punteggio minimo per ogni prova */
#define MAX_PUN  10      /* punteggio massimo per ogni prova */

main()
{
float prova1[MAX_CONC], prova2[MAX_CONC], totale[MAX_CONC];
int i, n;

do {
  printf("\nNumero concorrenti: ");
  scanf("%d", &n);
}
while(n<1 || n>MAX_CONC);

/* Per ogni concorrente, richiesta punteggio nelle due prove */
for(i=0; i<n; i++) {
  printf("\nConcorrente n.%d \n", i+1);

  do {
    printf("Prima prova: ");
    scanf("%f", &prova1[i]);
  }
  while(prova1[i]<MIN_PUN || prova1[i]>MAX_PUN);

  do {
    printf("Seconda prova: ");
    scanf("%f", &prova2[i]);
  }
  while(prova2[i]<MIN_PUN || prova2[i]>MAX_PUN);
}

/* Calcolo media per concorrente */
for(i=0; i<n; i++)
  totale[i] = (prova1[i]+prova2[i])/2;

printf("\n      CLASSIFICA\n");
for(i=0; i<n; i++)
  printf("%f  %f  %f \n", prova1[i], prova2[i], totale[i]);
}
```

Listato 4.2 Esempio di utilizzo di un array

Non conoscendo a priori il numero di concorrenti che parteciperanno alle gare si fa l'ipotesi che comunque non siano più di 1000, valore che memorizziamo nella costante `MAX_CONC`. In conseguenza di ciò definiamo di lunghezza `MAX_CONC` gli array che conterranno i risultati: `prova1`, `prova2` e `totale`. Richiediamo all'utente a tempo di esecuzione il numero effettivo dei concorrenti e verifichiamo che non sia minore di 1 e maggiore di `MAX_CONC`:

```
do {
       printf("\nNumero concorrenti: ");
       scanf("%d", &n);
     }
     while(n<MIN_PUN || n>MAX_CONC);
```

In seguito richiediamo l'introduzione dei risultati della prima e della seconda prova di ogni concorrente, controllando che tale valutazione non sia minore di 1 e maggiore di 10, nel qual caso ripetiamo la richiesta. Abbiamo memorizzato in `MIN_PUN` e `MAX_PUN` i limiti inferiore e superiore del punteggio assegnabile, in maniera che, se questi venissero modificati, basterebbe intervenire sulle loro definizioni perché il programma continui a funzionare correttamente. Infine calcoliamo il punteggio totale e lo visualizziamo. Un esempio di esecuzione è mostrato in Figura 4.4.

✓ **NOTA**

Si noti che soltanto *n* elementi di ogni array vengono utilizzati veramente; quindi se, per esempio, i concorrenti sono dieci, si ha un utilizzo di memoria pari a solo il 10 per mille (valore attuale di `MAX_CONC`); in questo modo però il programma è più flessibile.

Esistono altre soluzioni più complesse che permettono di gestire la memoria dinamicamente (cioè di adattarla alle effettive esigenze del programma), anziché staticamente (cioè riservando a priori uno spazio): le vedremo in seguito.

```
Numero concorrenti: 3

Concorrente n.1

Prima prova: 8   Seconda prova: 7

Concorrente n.2

Prima prova: 5   Seconda prova: 9

Concorrente n.3

Prima prova: 8   Seconda prova: 8


      CLASSIFICA

8.000000  7.000000  7.500000
5.000000  9.000000  7.000000
8.000000  8.000000  8.000000
```

Figura 4.4 Esempio di esecuzione del programma del Listato 4.2

4.3 Inizializzazione di variabili

L'inizializzazione di una variabile può essere esplicitata direttamente al momento della sua dichiarazione, come con

```
int i = 0;
```

L'istruzione dichiara la variabile `i` di tipo intero e le assegna il valore zero. Di seguito alla dichiarazione di tipo possono essere definite e inizializzate più variabili:

```
int a = 50, b = 30, c, d = 333;
```

definisce le variabili intere `a`, `b`, `c` e `d`; inizializza `a` al valore 50, `b` a 30, `d` a 333, `c` non è inizializzata. Analogamente si possono assegnare valori agli altri tipi di variabili semplici:

```
float x = 567.8927;
     float y = 7e13;
     char  risposta = 's';
```

Per gli array l'inizializzazione è possibile solamente se sono stati dichiarati come `extern` o come `static`; quest'ultima classe di variabile verrà esaminata in seguito.

Le variabili `extern` sono quelle che vengono definite prima di `main`. L'inizializzazione si ottiene inserendo i valori tra parentesi graffe, separati da una virgola:

```
int voti[6] = {11, 18, 7, 15, 21, 9};
```

Il compilatore fa la scansione dei valori presenti tra parentesi graffe da sinistra verso destra e genera altrettanti assegnamenti consecutivi agli elementi del vettore, rispettando la loro posizione; dunque `voti[0]` assume il valore 11, `voti[1]` 18, `voti[2]` 7 ecc. Quando tutti gli elementi dell'array vengono inizializzati è possibile omettere l'indicazione del numero di elementi, e scrivere

```
int voti[] = {11, 18, 7, 15, 21, 9};
```

È infatti il compilatore stesso che conta i valori e di conseguenza determina la dimensione del vettore.

Gli array di caratteri, comunemente detti *stringhe*, possono essere inizializzati anche inserendo il loro contenuto tra doppi apici:

```
char frase[] = "Analisi, requisiti";
```

4.4 Matrici

Nei paragrafi precedenti abbiamo trattato i vettori, detti anche matrici monodimensionali. Per la memorizzazione abbiamo usato una variabile di tipo array dichiarandone il numero di componenti, per esempio:

```
int vet[3];
```

Per accedere direttamente a ciascuno degli elementi del vettore si è utilizzato un indice che varia da zero a $n-1$. Nell'esempio n è uguale a 3.

In una matrice bidimensionale i dati sono organizzati per righe e per colonne, come se fossero inseriti in una tabella. Per la memorizzazione si utilizza una variabile di tipo array specificando il numero di componenti per ciascuna delle due dimensioni che la costituiscono:

```
int mat[4][3];
```

La variabile strutturata `mat` che abbiamo dichiarato contiene 4 righe e 3 colonne per un totale di dodici elementi; per accedere a ciascuno di essi si utilizzano due indici: il primo specifica la riga il secondo la colonna. Gli indici variano rispettivamente tra 0 e $r-1$ e tra 0 e $c-1$, dove r e c sono il numero di righe e il numero di colonne. Abbiamo cioè

```
mat[0][0]  mat[0][1]  mat[0][2]
     mat[1][0]  mat[1][1]  mat[1][2]
     mat[2][0]  mat[2][1]  mat[2][2]
     mat[3][0]  mat[3][1]  mat[3][2]
```

Per esempio, `mat[1][2]` fa riferimento all'elemento presente nella seconda riga della terza colonna. Ogni colonna della matrice bidimensionale non è altro che un vettore.

Il formato generale della dichiarazione degli array multidimensionali è il seguente:

```
tipo nome[dimensione1][dimensione2]...[dimensioneN];
```

Per esempio, al fine di memorizzare i ricavi ottenuti dalla vendita di 10 prodotti in 5 punti vendita nei dodici mesi dell'anno, potremmo utilizzare la matrice tridimensionale `marketing` così dichiarata:

```
int marketing[10][5][12]
```

Scriviamo ora un programma che richiede all'utente i valori da inserire, li memorizza nella matrice bidimensionale `mat` e la visualizza (Listato 4.3).

```
/* Caricamento di una matrice */

#include <stdio.h>

int mat[4][3];

main()
{
int i, j;

printf("\n \n CARICAMENTO DELLA MATRICE \n \n");
for(i=0; i<4; i++)
  for(j=0; j<3; j++) {
    printf("Inserisci linea %d colonna %d val: ", i, j);
    scanf("%d", &mat[i][j]);
  };

/* Visualizzazione */
for(i=0; i<4; i++) {
  printf("\n");
  for(j=0; j<3; j++)
    printf("%5d", mat[i][j]);
}
}
```

Listato 4.3 Esempio di utilizzo di un array bidimensionale

Per effettuare il caricamento dei dati nella matrice utilizziamo due cicli, uno più esterno che mediante la variabile `i` fa la scansione delle righe da zero a 3 (4−1) e un altro che percorre, per mezzo della variabile `j`, le colonne da zero a 2 (3−1):

```
for(i=0; i<4; i++)
       for(j=0; j<3; j++) {
         printf("Inserisci linea %d colonna %d val:", i, j);
         scanf("%d", &mat[i][j]);
       };
```

Viene riempita tutta la prima riga poi la seconda e così via; se l'utente passa i valori 2, 55, 12, 98, 34... essi verranno inseriti negli elementi `mat[0][0]`,`mat[0][1]`,`mat[0][2]`,`mat[1][0]`,`mat[1][1]`...

Si può ottenere il caricamento per colonne invertendo semplicemente i due cicli:

```
for(j=0; j<3; j++)
       for(i=0; i<4; i++) {
         printf("Inserisci linea %d colonna %d val:", i, j);
         scanf("%d", &mat[i][j]);
       };
```

Il numero totale d'iterazioni è sempre uguale a 12 e gli indici `j` e `i` fanno la scansione delle colonne e delle righe della matrice senza fuoriuscire dai margini. La visualizzazione è ancora una volta ottenuta con due cicli `for` annidati uno nell'altro.

Nel programma precedente le dimensioni della matrice erano fissate a priori: modifichiamolo in modo da far decidere all'utente il numero delle righe e delle colonne, come nel Listato 4.4.

```
/* Caricamento di una matrice
   le cui dimensioni vengono decise dall'utente */

#include <stdio.h>

#define MAXLINEE 100
#define MAXCOLONNE 100
int mat[MAXLINEE][MAXCOLONNE];

main()
{
int n, m;
int i, j;

/* Richiesta delle dimensioni */
do {
  printf("\nNumero di linee: ");
  scanf("%d", &n);
}
while((n>=MAXLINEE) || (n<1));

do {
  printf("Numero di colonne: ");
  scanf("%d", &m);
}
while((m>=MAXCOLONNE) || (m<1));

printf("\n \n CARICAMENTO DELLA MATRICE \n \n");
for(i=0; i<n; i++)
  for(j=0; j<m; j++) {
    printf("Inserisci linea %d colonna %d val:", i, j);
    scanf("%d", &mat[i][j]);
  };

/* Visualizzazione */
for(i=0; i<n; i++) {
  printf("\n");
  for(j=0; j<m; j++)
    printf("%5d", mat[i][j]);
}
}
```

Listato 4.4 Inizializzazione di una matrice bidimensionale, seconda versione

La matrice viene definita con un massimo numero di linee e di colonne:

```
int mat[MAXLINEE][MAXCOLONNE];
```

dove `MAXLINEE` e `MAXCOLONNE` sono due costanti che abbiamo dichiarato precedentemente, il cui valore deve essere scelto in relazione alle massime dimensioni. Successivamente si richiede l'inserimento del valore di *n*, numero di linee che realmente verranno riempite:

```
do {
      printf("Numero di linee: ");
       scanf("%d", &n);
```

```
    }
    while((n>=MAXLINEE) || (n<1));
```

L’istruzione `scanf` viene inserita in un ciclo `do-while` in modo che se *n* è maggiore del numero di linee che costituiscono la matrice o è minore di 1 il valore non viene accettato e la richiesta viene ripetuta. Analogamente si procede per le colonne.

4.5 Esempi con le matrici

Passiamo adesso a un problema più complesso: date due matrici `mat1[N][P]`, `mat2[P][M]` calcolare la matrice prodotto in cui ogni elemento è dato da:

$$pmat[i][j] = \sum_{k=1}^{P} \texttt{mat1[i][k]*mat2[k][j]}$$

`per i=1..N, j=1..M`

Il prodotto così definito si può ottenere soltanto se il numero di colonne della prima matrice (*P*) è uguale al numero di righe della seconda. La matrice `pmat` è dunque costituita da *N* righe e *M* colonne.

Consideriamo le matrici di Figura 4.5: l’elemento [2][4] della matrice prodotto è dato da

```
    pmat[2][4] = mat1[2][0]*mat2[0][4] +
                 mat1[2][1]*mat2[1][4] +
                 mat1[2][2]*mat2[2][4]
```

ossia

`pmat[2][4] = 5*3 + 2*4 + 0*5 = 23`

Venendo dunque al programma richiesto (Listato 4.5), in primo luogo si devono caricare i dati delle due matrici.

```
/* Calcolo del prodotto di due matrici */

#include <stdio.h>

#define N 4
#define P 3
#define M 5

int mat1[N][P];         /* prima matrice */
int mat2[P][M];         /* seconda matrice */
int pmat[N][M];         /* matrice prodotto */

main()
{
int i, j, k;

printf("\n \n CARICAMENTO DELLA PRIMA MATRICE \n \n");
for(i=0; i<N; i++)
  for(j=0; j<P; j++) {
    printf("Inserisci linea %d colonna %d val:", i, j);
    scanf("%d", &mat1[i][j]);
  };

printf("\n \n CARICAMENTO DELLA SECONDA MATRICE \n \n");
for(i=0; i<P; i++)
  for(j=0; j<M; j++) {
    printf("Inserisci linea %d colonna %d val:", i, j);
```

```
    scanf("%d", &mat2[i][j]);
  };

/* Calcolo del prodotto */
for(i=0; i<N; i++)
  for(j=0; j<M; j++) {
    pmat[i][j] = 0;
    for(k=0; k<P; k++)
      pmat[i][j] = pmat[i][j] + mat1[i][k] * mat2[k][j];
  };

printf("\n \n PRIMA MATRICE \n ");
for(i=0; i<N; i++) {
  printf("\n");
  for(j=0; j<P; j++)
    printf("%5d", mat1[i][j]);
}

printf("\n \n SECONDA MATRICE \n ");
for(i=0; i<P; i++) {
  printf("\n");
  for(j=0; j<M; j++)
    printf("%5d", mat2[i][j]);
}

printf("\n \n MATRICE PRODOTTO \n ");
for(i=0; i<N; i++) {
  printf("\n");
  for(j=0; j<M; j++)
    printf("%5d", pmat[i][j]);
}
}
```

Listato 4.5 Calcolo del prodotto tra matrici

Per ottenere il valore dell'elemento `i, j` della matrice prodotto lo si inizializza a zero:

```
pmat[i][j] = 0;
```

Successivamente, con un ciclo che fa la scansione della riga `i` di `mat1` e della colonna `j` di `mat2`, si accumula in `pmat[i][j]` la sommatoria dei prodotti dei corrispondenti elementi di `mat1` e `mat2`:

```
for(k=0; k<P; k++)
       pmat[i][j] = pmat[i][j] + mat1[i][k] * mat2[k][j];
```

La variabile `k` permette di scorrere contemporaneamente la linea `i` di `mat1` e la colonna `j` di `mat2`; il suo valore varia da 0 a P. Il procedimento appena visto va ripetuto per ognuno degli elementi della matrice prodotto:

```
for(i=0; i<N; i++)
       for(j=0; j<M; j++) {
         pmat[i][j] = 0;
         for(k=0; k<P; k++)
           pmat[i][j] = pmat[i][j] + mat1[i][k] * mat2[k][j];
       };
```

I due cicli `for` fissano a ogni iterazione una certa riga di `mat1` e di `pmat` e una certa colonna di `mat2` e di `pmat`.

Riportiamo in Figura 4.5 un esempio di esecuzione del programma.

```
 CARICAMENTO DELLA PRIMA MATRICE
```

```
Inserisci linea 0 colonna 0 val:1
Inserisci linea 0 colonna 1 val:0
Inserisci linea 0 colonna 2 val:0
Inserisci linea 1 colonna 0 val:22
Inserisci linea 1 colonna 1 val:-6
Inserisci linea 1 colonna 2 val:3
Inserisci linea 2 colonna 0 val:5
Inserisci linea 2 colonna 1 val:2
Inserisci linea 2 colonna 2 val:0
Inserisci linea 3 colonna 0 val:11
Inserisci linea 3 colonna 1 val:4
Inserisci linea 3 colonna 2 val:7


 CARICAMENTO DELLA SECONDA MATRICE

Inserisci linea 0 colonna 0 val:2
Inserisci linea 0 colonna 1 val:0
Inserisci linea 0 colonna 2 val:4
Inserisci linea 0 colonna 3 val:0
Inserisci linea 0 colonna 4 val:3
Inserisci linea 1 colonna 0 val:0
Inserisci linea 1 colonna 1 val:1
Inserisci linea 1 colonna 2 val:5
Inserisci linea 1 colonna 3 val:1
Inserisci linea 1 colonna 4 val:4
Inserisci linea 2 colonna 0 val:21
Inserisci linea 2 colonna 1 val:1
Inserisci linea 2 colonna 2 val:2
Inserisci linea 2 colonna 3 val:2
Inserisci linea 2 colonna 4 val:5


 PRIMA MATRICE

    1    0    0
   22   -6    3
    5    2    0
   11    4    7

 SECONDA MATRICE

    2    0    4    0    3
    0    1    5    1    4
   21    1    2    2    5

 MATRICE PRODOTTO

    2    0    4    0    3
  107   -3   64    0   57
   10    2   30    2   23
  169   11   78   18   84
```

Figura 4.5 Esempio di esecuzione del programma del Listato 4.5

4.6 Esercizi

* 1. Scrivere un programma che, inizializzati in due vettori `a` e `b` della stessa lunghezza *n* valori interi, calcoli la somma incrociata degli elementi: `a[1] + b[n]`, `a[2] + b[n-1]`, ... la memorizzi nel vettore `c` e visualizzi quindi `a`, `b` e `c`.

* 2. Modificare il programma, esaminato nel presente capitolo, che determina il maggiore, il minore e la media degli elementi di un array in modo che vengano diminuiti in media il numero di confronti effettuati nel ciclo durante l'esecuzione.

3. Scrivere un programma che inizializzi e quindi visualizzi un vettore con i valori alternati 0, 1, 0, 1, 0, 1, 0, 1, … Ripetere l'esercizio con i valori 0, −3, 6, −9, 12, −15, 18, −21, ...

4. Scrivere un programma che, letti gli elementi di un vettore `v1` e un numero `k`, determini l'elemento di `v1` più prossimo a `k`.

5. Scrivere un programma che, letti gli elementi di due vettori `v1` e `v2` di lunghezza 5, determini il vettore `w` di lunghezza 10 ottenuto alternando gli elementi di `v1` e `v2`. Visualizzare `v1`, `v2` e `w`.
Per esempio: se `v1` e `v2` sono i vettori di caratteri

v1	B	N	S	I	O

V2	E	I	S	M	!

si deve ottenere il vettore

W	B	E	N	I	S	S	I	M	O	!

6. Scrivere un programma che, letti gli elementi di due vettori `v1` e `v2` di lunghezza *n*, inizializzi un terzo vettore `w` di lunghezza *n* con i valori

```
w(i) = 1          se v1(i)>v2(i);
w(i) = 0          se v1(i)=v2(i);
w(i) = -1         se v1(i)<v2(i).
```

Visualizzare quindi `v1`, `v2` e `w`.

7. Scrivere un programma che, inizializzato un vettore di `char` con una stringa di lettere dell'alfabeto e punteggiatura, visualizzi il numero complessivo delle vocali e delle consonanti del vettore.

8. Scrivere un programma di inizializzazione che richiesto un elemento controlli, prima di inserirlo nel vettore, se è già presente, nel qual caso chieda che l'elemento sia digitato di nuovo.

9. Scrivere un programma che inizializzi e quindi visualizzi una matrice di `int` in cui ciascun elemento è dato dalla somma dei propri indici.

10. [*Matrici simmetriche*] Una matrice quadrata $n \times n$ di un tipo qualsiasi si dice simmetrica se gli elementi simmetrici rispetto alla diagonale principale (dal vertice alto sinistro al vertice basso destro) sono due a due uguali. Scrivere un programma che, letta una matrice quadrata di interi, controlli se è simmetrica.

11. Scrivere un programma che, inizializzata una matrice $n \times n$, visualizzi la matrice che si ottiene da quella data scambiando le righe con le colonne.

* 12. Modificare il programma che calcola il prodotto di due matrici bidimensionali esaminato nel presente capitolo, in modo che sia l'utente a scegliere le dimensioni degli array. Il programma deve verificare la correttezza delle dimensioni inserite.

13. Scrivere un programma che, letta una matrice di interi o reali, individui la colonna con somma degli elementi massima.

* 14. Scrivere un programma che richieda all'utente i voti delle otto prove sostenute durante l'anno da diciotto studenti di una classe e calcoli la media di ogni studente, la media di ogni prova e la media globale. Il programma dovrà infine visualizzare l'intera matrice e la media globale. [*Suggerimento*: si utilizzi una matrice di 19 linee e 9 colonne dove nelle prime otto vengono memorizzati in ciascuna linea i voti di uno studente e nella nona la rispettiva media; nella diciannovesima linea viene invece memorizzata la media per prova.]

15. Memorizzare in un array tridimensionale i numeri estratti al gioco del lotto su tutte le ruote per dieci estrazioni consecutive. Verificare su quali ruote e in quali estrazioni si ripete un certo numero passato in ingresso dall'utente.

5.1 Introduzione

Questo capitolo ci permette di fare pratica di programmazione utilizzando gli strumenti del linguaggio introdotti finora. A una prima lettura possono essere saltati senza che ciò pregiudichi l'apprendimento degli argomenti seguenti, nel caso il lettore voglia continuare ad appropriarsi rapidamente delle possibilità offerte dal linguaggio.
È esperienza comune che nella gestione dei più svariati insiemi di dati (vettori o matrici, ma più in generale anche archivi cartacei, listini prezzi, voci di un'enciclopedia o addirittura semplici carte da gioco) sia spesso necessario: stabilire se un elemento è o no presente nell'insieme, ordinare l'insieme in un determinato modo (in genere in maniera crescente o decrescente), unire (fondere) due o più insiemi in un unico insieme evitando possibili duplicazioni. Queste tre attività, che in informatica vengono indicate rispettivamente con i termini di *ricerca*, *ordinamento* e *fusione* oppure con i loro equivalenti inglesi *search*, *sort* e *merge*, sono estremamente frequenti e svolgono un ruolo della massima importanza in tutti i possibili impieghi degli elaboratori. È stato per esempio stimato che l'esecuzione dei soli programmi di ordinamento rappresenti oltre il 30% del lavoro svolto dai computer. È quindi ovvio come sia della massima importanza disporre di programmi che svolgano questi compiti nel minor tempo possibile.

5.2 Ricerca completa

Un primo algoritmo per determinare se un valore è presente all'interno di un array, applicabile anche a sequenze non ordinate, è quello comunemente detto di *ricerca completa*, che opera una scansione sequenziale degli elementi del vettore confrontandoli con il valore ricercato. Nel momento in cui tale verifica dà esito positivo la scansione ha termine e viene restituito l'indice dell'elemento all'interno dell'array stesso.
Per determinare che il valore non è presente, il procedimento (Listato 5.1) deve controllare uno a uno tutti gli elementi fino all'ultimo, prima di poter sentenziare il fallimento della ricerca. L'array che conterrà la sequenza è `vet` formato da `MAX_ELE` elementi.

```
/* Ricerca sequenziale di un valore nel vettore */

#include <stdio.h>
#define MAX_ELE 1000   /* massimo numero di elementi */

main()
{
char vet[MAX_ELE];
int i, n;
char c;

/* Immissione lunghezza della sequenza */
do {
  printf("\nNumero elementi: ");
  scanf("%d", &n);
}
while(n<1 || n>MAX_ELE);

/* Immissione elementi della sequenza */
for(i=0; i<n; i++) {
  printf("\nImmettere carattere n.%d: ",i);
  scanf("%1s", &vet[i]);
}

printf("Elemento da ricercare: ");
scanf("%1s", &c);
```

```
/* Ricerca sequenziale */
i = 0;
while(c!=vet[i] && i<n-1)  ++i;
if(c==vet[i])
  printf("\nElemento %c presente in posizione %d\n",c,i);
else
  printf("\nElemento non presente!\n");
}
```

Listato 5.1 Ricerca completa

Il programma presenta le solite fasi di richiesta e relativa immissione del numero degli elementi della sequenza e dei valori che la compongono. Successivamente l'utente inserisce il carattere da ricercare, che viene memorizzato nella variabile `c`. La ricerca parte dal primo elemento dell'array (quello con indice zero) e prosegue fintantoché il confronto fra `c` e `vet[i]` dà esito negativo e contemporaneamente `i` è minore di `n-1`:

```
while(c!=vet[i] && i<n-1)  ++i;
```

Il corpo del ciclo è costituito dal semplice incremento di `i`. L'iterazione termina in tre casi:

1. 1. `c` è uguale a `vet[i]` e `i` è minore di `n-1`;
2. 2. `c` è diverso da `vet[i]` e `i` è uguale a `n-1`;
3. 3. `c` è uguale a `vet[i]` e `i` è uguale a `n-1`.

In ogni caso `i` ha un valore minore o uguale a `n-1`, è dunque all'interno dei limiti di esistenza dell'array. L'`if` successivo determinerà se è terminato perché `c` è risultato essere uguale a `vet[i]`:

```
if(c==vet[i])
```

Esistono molte altre soluzioni al problema. Per esempio si potrebbe adottare un costrutto `while` ancora più sintetico, come il seguente:

```
while(c!=v et[i] && i++<n-1)
        ;
```

dove il corpo del ciclo non è esplicitato in quanto l'incremento di `i` avviene all'interno dell'espressione di controllo. Si noti però che, in questo caso, al termine delle iterazioni `i` ha un valore maggiorato di uno rispetto alla condizione che ha bloccato il ciclo, e di questo bisognerà tener conto nel prosieguo del programma ▬.

5.3 Ordinamenti

Un altro, fondamentale problema dell'informatica spesso collegato con la ricerca è quello che consiste nell'ordinare un vettore disponendo i suoi elementi in ordine crescente o decrescente. Esistono numerosi algoritmi che consentono di ordinare un array; uno dei più famosi è quello comunemente detto *bubblesort*. Osserviamo una sua versione in C che ordina un array in modo crescente. L'array è identificato da `vet` e ha *n* elementi:

```
for(j=0;j<n-1; j++)
       for(i=0; i<n-1; i++)
         if(vet[i]>vet[i+1])
           {aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux;}
```

Nel ciclo più interno gli elementi adiacenti vengono confrontati: se `vet[i]` risulta maggiore di `vet[i+1]` si effettua lo scambio tra i loro valori. Per quest'ultima operazione si ha la necessità di una variabile di appoggio, che nell'esempio è `aux`. Il ciclo si ripete finché tutti gli elementi sono stati confrontati, quindi fino a quando `i` è minore di `n-1`, perché il confronto viene fatto tra `vet[i]` e `vet[i+1]`.

Questa serie di confronti non è in generale sufficiente a ordinare l'array. La sicurezza dell'ordinamento è data dalla sua ripetizione per $n-1$ volte; nell'esempio ciò si ottiene con un `for` più esterno controllato dalla variabile `j` che varia da 0 a $n-1$ (Figura 5.1).

```
vet[0] = 9
vet[1] = 18
vet[2] = 7
vet[3] = 15
vet[4] = 21
vet[5] = 11
```

	9	9	9	9	9	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7
	18	7	7	7	7	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
	7	18	15	15	15	15	15	15	15	15	15	15	11	11	11	11	11	11	11	11	11	11	11	11	11
	15	15	18	18	18	18	18	18	11	11	11	11	15	15	15	15	15	15	15	15	15	15	15	15	15
	21	21	21	21	11	11	11	11	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18	18
	11	11	11	11	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21	21
i	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
j	0					1					2					3					4				

Figura 5.1 Esempio di ordinamento con l'algoritmo di bubblesort

In realtà il numero di volte per cui il ciclo interno va ripetuto dipende da quanto è disordinata la sequenza di valori iniziali. Per esempio, l'ordinamento di un array di partenza con valori 10, 12, 100, 50, 200, 315 ha bisogno di un solo scambio, che viene effettuato per `i=2` e `j=0`; dunque tutti i cicli successivi sono inutili. A questo proposito si provi a ricostruire i passaggi della Figura 5.1 con questi valori di partenza.

Si può dedurre che l'array è ordinato e cessare l'esecuzione delle iterazioni quando un intero ciclo interno non ha dato luogo ad alcuno scambio di valori tra `vet[i]` e `vet[i+1]`:

```
do {
       k = 0;
       for(i=0; i<n-1; i++)
         if(vet[i]>vet[i+1])
           {aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux; k=1;}
     }
     while(k==1);
```

Una prima ottimizzazione dell'algoritmo si ottiene interrompendo il ciclo esterno la prima volta che per un'intera iterazione del ciclo interno la clausola `if` non ha dato esito positivo.

Nel ciclo esterno la variabile `k` viene inizializzata a zero: se almeno un confronto del ciclo piccolo dà esito positivo, a `k` viene assegnato il valore uno. In pratica la variabile `k` è utilizzata come *flag* (bandiera): se il suo valore è 1 il ciclo deve essere ripetuto, altrimenti no.

Nel caso dell'array di partenza di Figura 5.1, l'adozione dell'ultimo algoritmo fa risparmiare un ciclo esterno (cinque cicli interni) rispetto al precedente. La prima volta che l'esecuzione del ciclo esterno non dà esito a scambi corrisponde al valore di `j` uguale a 3, `k` rimane a valore zero e le iterazioni hanno termine. Si provi con valori iniziali meno disordinati per verificare l'ulteriore guadagno in tempo d'esecuzione.

Osservando ancora una volta la Figura 5.1 si nota che a ogni incremento di `j`, variabile che controlla il ciclo esterno, almeno gli ultimi `j+1` elementi sono ordinati. Il fatto è valido in generale poiché il ciclo interno sposta di volta in volta l'elemento più pesante verso il basso. Dall'ultima osservazione possiamo ricavare un'ulteriore ottimizzazione dell'algoritmo:

```
do {
       k = 0;
       for(i=0; i<n-1; i++)
         if(vet[i]>vet[i+1])
           {aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux; k=1;}
       --n;
     }
```

```
	while(k==1);
```

In tale ottimizzazione, a ogni nuova ripetizione del ciclo esterno viene decrementato il valore limite del ciclo interno, in modo da diminuire di uno, di volta in volta, il numero di confronti effettuati. Ma è ancora possibile un'altra ottimizzazione:

```
p = n;
     do {
       k = 0;
       for(i=0; i<n-1; i++)
         if(vet[i]>vet[i+1]) {
           aux=vet[i]; vet[i]=vet[i+1]; vet[i+1]=aux;
            k = 1; p = i+1;
         }
     n = p;
     }
     while(k==1);
```

Il numero dei confronti effettuati dal ciclo interno si interrompe lì dove la volta precedente si è avuto l'ultimo scambio, come si osserva dal confronto tra le Figure 5.1 e 5.2.

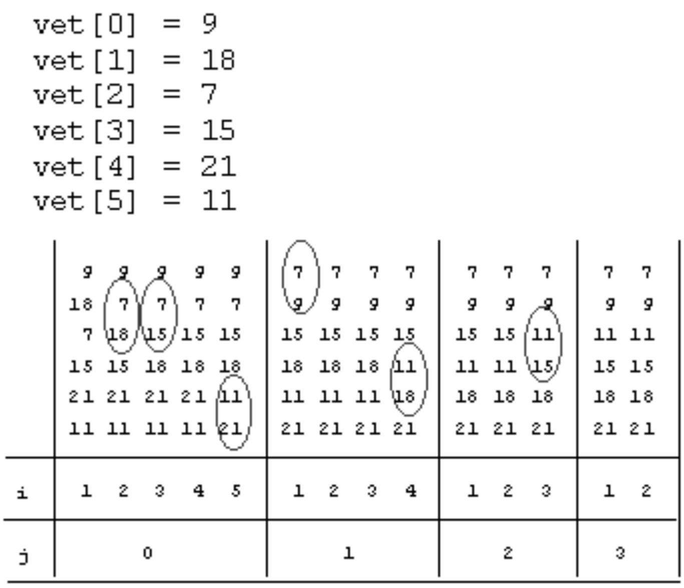


Figura 5.2 Esempio di ordinamento con l'algoritmo di *bubblesort* ottimizzato

5.4 Ricerca binaria

Quando l'array risulta ordinato la ricerca di un valore al suo interno può avvenire mediante criteri particolari, uno dei quali è la ricerca detta *binaria* o *dicotomica*.

```
/* Ricerca binaria */

#include <stdio.h>

main()
{
```

```
char vet[6];           /* array contenente i caratteri immessi */
int i,n,k,p;
char aux;              /* variabile di appoggio per lo scambio */
char ele;              /* elemento da ricercare */
int basso, alto, pos;  /* usati per la ricerca binaria */

/* Immissione caratteri */
n = 6;
for(i=0;i<=n-1; i++) {
  printf("vet %d° elemento: ", i+1);
  scanf("%1s", &vet[i]);
}

/* ordinamento ottimizzato */
p = n;
do {
  k = 0;
  for(i=0; i<n-1; i++) {
    if(vet[i]>vet[i+1]) {
      aux = vet[i]; vet[i] = vet[i+1]; vet[i+1] = aux;
      k = 1; p = i+1;
    }
  }
n = p;
}
while(k==1);

printf("\nElemento da ricercare: ");
scanf("%1s", &ele);

/* ricerca binaria */
n = 6;
alto = 0; basso = n-1; pos = -1;
do {
  i = (alto+basso)/2;
  if(vet[i]==ele)  pos = i;
  else
    if(vet[i]<ele)
      alto = i+1;
    else
      basso = i-1;
}
while(alto<=basso && pos==-1);

if(pos != -1)
  printf("\nElemento %c presente in posizione %d\n",ele,pos);
else
  printf("\nElemento non presente! %d\n", pos);
}
```

Listato 5.2 Programma completo di immissione, ordinamento e ricerca

Nel Listato 5.2, dopo l'immissione dei valori del vettore, il loro ordinamento con bubblesort e l'accettazione dell'elemento da cercare, abbiamo i comandi della ricerca binaria vera e propria:

```
/* ricerca binaria */
     alto = 0; basso = n-1; pos = -1;
     do {
```

```
    i = (alto+basso)/2;
    if(vet[i]==ele) pos=i;
    else
      if(vet[i]<ele)
        alto = i+1;
      else
        basso = i-1;
  }
  while(alto<=basso && pos==-1);
```

Si confronta il valore da ricercare, che è memorizzato nella variabile `ele`, con l'elemento intermedio dell'array. L'indice `i` di tale elemento lo si calcola sommando l'indice inferiore dell'array (`0`), memorizzato nella variabile `alto`, con quello superiore (`n-1`), memorizzato nella variabile `basso`, e dividendolo per due. Essendo l'array ordinato si possono presentare tre casi:

1. 1. `vet[i]` è uguale a `ele`, la ricerca è finita positivamente, si memorizza l'indice dell'elemento in `pos` e il ciclo di ricerca ha termine;
2. 2. `vet[i]` è minore di `ele`, la ricerca continua tra i valori maggiori di `vet[i]` che sono memorizzati negli elementi con indice compreso tra `i+1` e `basso`, per cui si assegna ad `alto` il valore `i+1`. Se non si sono già esaminati tutti gli elementi del vettore (`alto` non è minore o uguale a `basso`) la ricerca continua assegnando ancora una volta a `i` il valore `(basso+alto)/2`;
3. 3. `vet[i]` è maggiore di `ele`, la ricerca continua tra i valori minori di `vet[i]` che sono memorizzati negli elementi con indice compreso tra `alto` e `i-1`, per cui si assegna a `basso` il valore `i-1`. Se non si sono già esaminati tutti gli elementi del vettore (`alto` non è minore di `basso`) la ricerca continua assegnando ancora una volta a `i` il valore `(basso+alto)/2`.

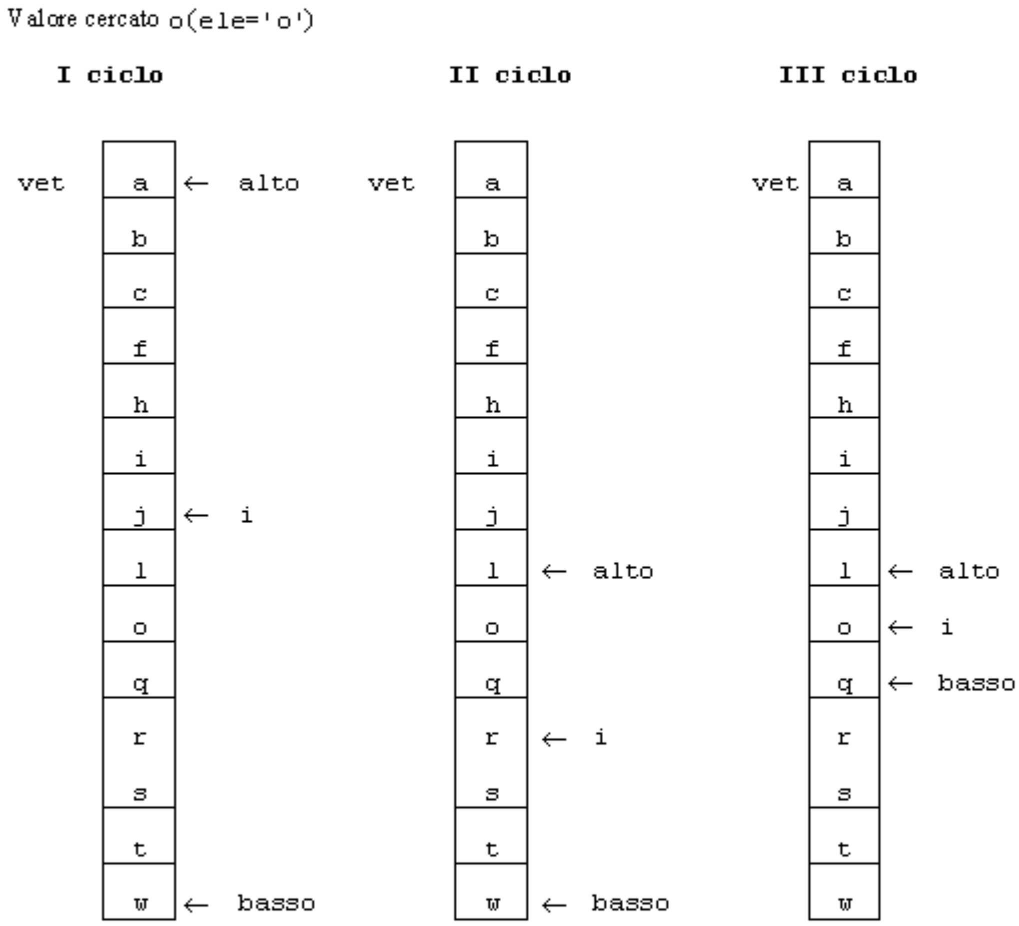


Figura 5.3 Esempio di ricerca binaria di `ele='o'`.

Nella Figura 5.3 si osserva il mutare dei valori di `alto`, `basso` e `i` fino al reperimento del valore desiderato (“`o`”). Il numero di cicli e corrispondenti confronti effettuati è risultato uguale a tre ▬, mentre se avessimo utilizzato la ricerca sequenziale avremmo avuto nove iterazioni. La ricerca sequenziale esegue nel caso più fortunato – quello in cui l’elemento cercato è proprio il primo – un unico confronto; nel caso più sfortunato – quello in cui l’elemento cercato è invece l’ultimo – esegue *n* confronti. Si ha quindi che la ricerca sequenziale effettua in media (*n*+1)/2 confronti.
La ricerca binaria offre delle prestazioni indubbiamente migliori: al massimo esegue un numero di confronti pari al logaritmo in base due di *n*. Questo implica che nel caso in cui *n* sia uguale a 1000 per la ricerca sequenziale si hanno in media 500 confronti, per quella binaria al massimo 10. Poiché, come per l’ordinamento, il tempo impiegato per eseguire il programma è direttamente proporzionale al numero dei confronti effettuati, è chiaro come la ricerca binaria abbia tempi di risposta mediamente molto migliori della ricerca sequenziale ▬.
Osserviamo, tuttavia, che mentre si può effettuare la ricerca sequenziale su qualsiasi vettore, per la ricerca binaria è necessario disporre di un vettore ordinato ▬, così che non sempre risulta possibile applicare tale algoritmo ▬.

5.5 Fusione

Un altro algoritmo interessante è quello che partendo da due array monodimensionali ordinati ne ricava un terzo, anch’esso ordinato. I due array possono essere di lunghezza qualsiasi e in generale non uguale. Il programma del Listato 5.3 richiede all’utente l’immissione della lunghezza di ognuna delle due sequenze e gli elementi che le compongono. Successivamente ordina le sequenze ed effettua la fusione (*merge*) di una nell’altra, memorizzando il risultato in un array a parte.

```
/* Fusione di due sequenze ordinate */

#include <stdio.h>
#define MAX_ELE 1000

main()
{
char vet1[MAX_ELE];       /* prima sequenza */
char vet2[MAX_ELE];       /* seconda sequenza */
char vet3[MAX_ELE*2];     /* merge */

int n;                    /* lunghezza prima sequenza */
int m;                    /* lunghezza seconda sequenza */

char aux;                /* variabile di appoggio per lo scambio */

int i, j, k, p, n1, m1;

do {
  printf("Lunghezza prima sequenza: ");
  scanf("%d", &n);
}
while(n<1 || n>MAX_ELE);

/* caricamento prima sequenza */
for(i = 0;i <= n-1; i++) {
  printf("vet1 %d° elemento: ",i+1);
  scanf("%1s", &vet1[i]);
}

do {
  printf("Lunghezza seconda sequenza: ");
  scanf("%d", &m);
}
while(m<1 || m>MAX_ELE);
```

```
/* caricamento seconda sequenza */
for(i=0; i<=m-1; i++) {
  printf("vet2 %d° elemento: ",i+1);
  scanf("%1s", &vet2[i]);
}

/* ordinamento prima sequenza */
p = n; n1 = n;
do {
  k = 0;
  for(i = 0; i < n1-1; i++) {
    if(vet1[i]> vet1[i+1]) {
      aux = vet1[i]; vet1[i] = vet1[i+1]; vet1[i+1] = aux;
      k = 1; p = i+1;
    }
  }
n1 = p;
}
while(k==1);

/* ordinamento seconda sequenza */
p = m; m1 = m;
do {
  k = 0;
  for(i=0; i<m1 - 1; i++) {
    if(vet2[i]>vet2[i+1]) {
      aux = vet2[i]; vet2[i] = vet2[i+1]; vet2[i+1] = aux;
      k = 1; p = i+1;
    }
  }
m1 = p;
}
while(k==1);

/* fusione delle due sequenze (merge) */
i = 0; j = 0; k = 0;
do {
  if(vet1[i]<=vet2[j])
    vet3[k++] = vet1[i++];
  else
    vet3[k++] = vet2[j++];
}
while(i<n && j<m);

if(i<n)
  for(; i<n; vet3[k++] = vet1[i++])
    ;
else
  for(; j<m; vet3[k++] = vet2[j++])
    ;

/* visualizzazione della fusione */
for(i=0; i<k; i++)
  printf("\n%c", vet3[i]);
}
```

Listato 5.3 Fusione di due array

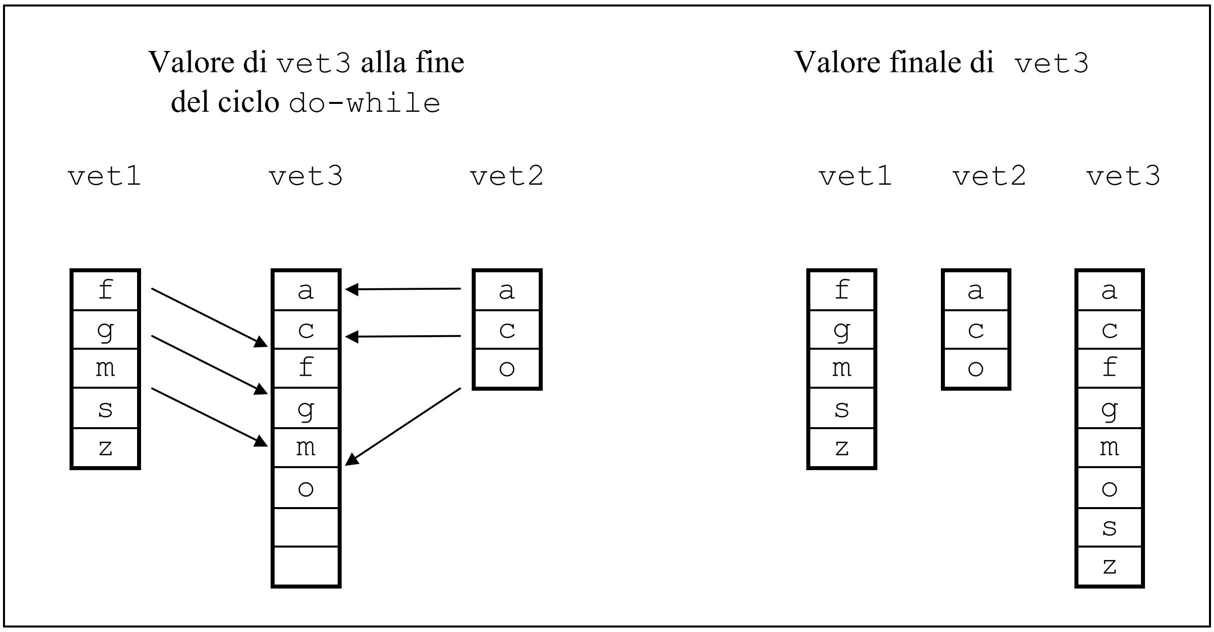


Figura 5.4 Risultato parziale e finale della fusione tra due vettori

In Figura 5.4 osserviamo il merge tra gli array ordinati `vet1` e `vet2` ordinati. L'operazione viene effettuata in due parti. La prima è data da:

```
i = 0; j = 0; k = 0;
     do {
       if(vet1[i]<=vet2[j])
         vet3[k++] = vet1[i++];
       else
         vet3[k++] = vet2[j++];
     }
     while(i<n && j<m);
```

Si controlla se l'*i*-esimo elemento di `vet1` è minore o uguale al *j*-esimo elemento di `vet2`, nel qual caso si aggiunge `vet1[i]` a `vet3` e si incrementa `i`. Nel caso contrario si aggiunge a `vet3` l'array `vet2[j]` e si incrementa `j`. In ogni caso si incrementa `k`, la variabile che indicizza `vet3`, perché si è aggiunto un elemento a `vet3`. Dal ciclo si esce quando `i` ha valore `n-1` o `j` ha valore `m-1`.

Si devono ancora aggiungere a `vet3` gli elementi di `vet1` (`j=m-1`) o di `vet2` (`i=n-1`) che non sono stati considerati. Nell'esempio precedente in `vet3` non ci sarebbero `s` e `z`. La seconda parte del merge ha proprio questo compito:

```
if(i<n)
       for(; i<n; vet3[k++] = vet1[i++])
         ;
     else
       for(; j<m; vet3[k++] = vet2[j++])
         ;
```

5.6 Esercizi

* 1. Scrivere un programma di ordinamento in senso decrescente .

* 2. Scrivere un programma che carichi una matrice bidimensionale di caratteri e successivamente ricerchi al suo interno un valore passato in ingresso dall'utente. Il programma restituisce quindi il numero di linea e di colonna relativo all'elemento cercato se questo è presente nella matrice, il messaggio `Elemento non presente` altrimenti.

3. Modificare il programma per la ricerca binaria in modo che visualizzi i singoli passi effettuati (cioè mostri i dati di Figura 5.3). Sperimentare il comportamento del programma con la ricerca dell'elemento 45 nel seguente vettore:

vet	21	33	40	41	45	50	60	66	72	81	88	89	91	93	99

4. Verificare, analogamente a quanto fatto in Figura 5.1, il comportamento della prima versione di bubblesort applicata al seguente vettore:

vet	3	31	1	23	41	5	0	66	2	8	88	9	91	19	99

5. Verificare il comportamento della versione ottimizzata di bubblesort applicata al vettore del precedente esercizio. Quanti cicli interni si sono risparmiati rispetto alla prima versione?

6. Calcolare il numero di confronti effettuati dall'algoritmo di ordinamento ingenuo applicato al vettore dell'Esercizio 4 e confrontarlo con quello di bubblesort.

7. Scrivere un programma che, richiesti i valori di un vettore ordinato in modo crescente, li inverta ottenendo un vettore decrescente. Si chiede di risolvere il problema utilizzando un solo ciclo.

8. Verificare il comportamento del programma di fusione applicato ai seguenti vettori:

vet1	3	31	41	43	44	45	80	
vet2	5	8	21	23	46	51	60	66

9. Modificare l'algoritmo di ricerca binaria nel caso il vettore sia ordinato in modo decrescente invece che crescente.

10. Se il vettore è ordinato la ricerca completa può essere migliorata in modo da diminuire in media il numero di confronti da effettuare: come? Modificare in questo senso il programma esaminato nel presente capitolo.

11. Scrivere un programma che, richiesti all'utente i primi $n-1$ elementi già ordinati di un vettore di dimensione n e un ulteriore elemento finale, inserisca quest'ultimo nella posizione corretta facendo *scivolare* verso il basso tutti gli elementi più grandi.

12. [*Insertion-sort*] Utilizzare l'algoritmo del precedente esercizio per scrivere un programma che ordini il vettore contemporaneamente all'inserimento dei dati da parte dell'utente.

13. Scrivere un programma che, richiesti all'utente i valori di una matrice, ne ordini tutte le colonne in senso crescente.

15. Scrivere un programma che, richiesti all'utente i valori di una matrice, ne ordini le righe in modo che il vettore i cui elementi corrispondono alla somma delle righe risulti ordinato in senso crescente.

6.1 Definizione

Una variabile di tipo `char` consente di memorizzare un singolo carattere. Molto spesso, però, è comodo poter trattare come una sola unità un insieme di caratteri alfanumerici, detto *stringa*; a questo scopo si possono utilizzare gli array di `char`. La linea di codice

```
char a[10];
```

dichiara un vettore costituito da dieci caratteri.

```
char frase[] = "Analisi, requisiti ";
```

dichiara invece l'array monodimensionale di caratteri `frase`, il cui numero di elementi è determinato dalla quantità di caratteri presenti tra doppi apici più uno, il carattere *null* (`\0`) che chiude la stinga (Figura 6.1).

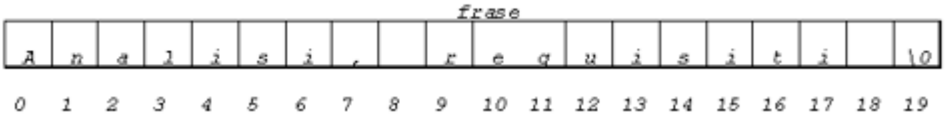


Figura 6.1 Contenuto dell'array `frase[]`.

Il carattere `\0` è il primo del codice ASCII, corrisponde alla notazione binaria 00000000 e non ha niente a che vedere con il carattere `0` che corrisponde a 00110000.

È importante osservare la differenza tra le due inizializzazioni:

```
char d = 'r';
     char b[] = "r";
```

La prima assegna alla variabile `d` di tipo `char` il valore `r`, la seconda assegna all'array `b[]` la sequenza di caratteri `r` e `\0`; in quest'ultimo caso si tratta effettivamente di una stringa. Naturalmente, quando si desidera far riferimento a un carattere si deve inserirlo tra apici singoli: per esempio

```
b[2] = 't';
```

assegna al terzo elemento dell'array `b` il carattere `t`.

Il carattere terminatore `\0` ci permette di trattare le stringhe senza conoscere a priori la dimensione.

Il programma del Listato 6.1 consente di verificare la corrispondenza tra ogni carattere presente in una stringa e il suo equivalente valore all'interno del codice ASCII, espresso nei sistema decimale e ottale.

```
/* Visualizzazione caratteri di una stringa */

#include <stdio.h>

char frase[] = "Analisi, requisiti ";

main()
{
int i=0;
while(frase[i]!='\0') {
  printf("%c = %d = %o \n", frase[i], frase[i], frase[i]);
  i++;
}
}
```

Listato 6.1 Visualizzazione di differenti rappresentazioni di caratteri

Il ciclo `while` permette di fare la scansione, uno a uno, dei caratteri della stringa. Viene controllato se il carattere in esame è `\0`, nel qual caso non ci sono più caratteri da esaminare e l'iterazione ha termine. L'istruzione `printf` visualizza a ogni ciclo un elemento dell'array, in tre formati differenti:

```
printf("%c = %d = %o \n", frase[i], frase[i], frase[i]);
```

Il primo formato, specificato da `%c`, indica il carattere ASCII stesso, il secondo e il terzo sono i suoi corrispondenti codici espressi nel sistema decimale (`%d`) e ottale (`%o`). Questo gioco di corrispondenze tra caratteri e numeri interi, definite dal codice ASCII, è sempre valido e offre grande libertà al programmatore.

L'esecuzione del programma dà il risultato:

```
A = 65   = 101
     n = 110 = 156
     a = 97  = 141
     l = 108 = 154
     i = 105 = 151
     s = 115 = 163
     i = 105 = 151
     , = 44  = 54
       = 32  = 40
     r = 114 = 162
     e = 101 = 145
     q = 113 = 161
     u = 117 = 165
     i = 105 = 151
     s = 115 = 163
     i = 105 = 151
     t = 116 = 164
     i = 105 = 151
       = 32  = 40
```

Comunque, se si desidera la visualizzazione dell'intera stringa, è possibile usare l'istruzione `printf` tramite la specifica del formato `%s`:

```
printf("%s", frase);
```

Tale istruzione, se inserita nel Listato 6.1, restituirebbe:

```
Analisi, requisiti
```

In questo caso è l'istruzione `printf` stessa che provvede a stampare carattere per carattere la stringa e a bloccarsi nel momento in cui identifica il carattere `\0`.

6.2 Esempi di uso delle stringhe

Il programma del Listato 6.2 copia il contenuto di una stringa in un'altra.

```
/* Copia di una stringa su un'altra */

#include <stdio.h>

char frase[] = "Analisi, requisiti ";

main()
{
int i;
char discorso[80];
for(i=0; (discorso[i]=frase[i])!='\0'; i++)
  ;
printf(" originale: %s \n copia:     %s \n", frase, discorso);
}
```

Listato 6.2 Copia di un array di caratteri

La variabile intera `i` viene utilizzata come indice degli array, per fare la scansione delle due stringhe carattere per carattere e per effettuare la copia. Nell'istruzione `for` viene inizializzato il valore di `i` a 0:

```
i = 0;
```

Successivamente il ciclo assegna a `discorso[0]` il valore di `frase[0]` e controlla che tale valore non sia uguale a `\0` (marca di fine stringa), nel qual caso il ciclo ha termine. A ogni nuova iterazione il valore di `i` viene incrementato di 1:

```
i++;
```

Viene quindi assegnato a `discorso[i]` il valore di `frase[i]`,

```
(discorso[i]=frase[i])
```

e controllato che il carattere in questione non sia `\0`:

```
(discorso[i]=frase[i])!='\0'
```

nel qual caso l'iterazione ha termine. Si noti che anche il carattere terminatore viene copiato nell'array `discorso`, che sarà così correttamente delimitato.

L'istruzione `printf` stampa la stringa di partenza e quella di arrivo, per cui l'esecuzione del programma visualizzerà:

```
originale: Analisi, requisiti
     copia:     Analisi, requisiti
```

✓ **NOTA**

Abbiamo evidenziato il fatto che il costrutto `for` non contiene alcuna istruzione esplicita all'interno del ciclo ponendo il punto e virgola nella riga successiva:

```
for(i=0; (discorso[i]=frase[i])!='\0'; i++)
  ;
```

Questa una notazione è utilizzata spesso in C. In realtà, come abbiamo visto esaminando il programma, qualcosa di concreto accade a ogni iterazione: un elemento di `frase` viene assegnato a `discorso`; l'operazione relativa è però contenuta (nascosta) all'interno di un'espressione della parte controllo del `for`. È chiaro che si poteva ottenere lo stesso risultato scrivendo:

```
for(i=0; frase[i]!='\0'; i++)
  discorso[i]=frase[i];
discorso[i]='\0';
```

Anche se questa notazione è più familiare, si esorta a utilizzare la precedente per sfruttare al meglio le caratteristiche del C.

Se si desidera copiare soltanto alcuni caratteri della prima stringa sulla seconda si deve modificare l'istruzione `for` del Listato 6.2. Scrivendo

```
for(i=0; ((discorso[i]=frase[i])!='\0') && (i<7); i++)
```

si è inserita la condizione `i<7` messa in AND (`&&`) con la verifica di fine stringa; in questo modo verranno copiati solamente i primi sette caratteri. L'istruzione `printf` visualizzerà allora:

```
originale: Analisi, requisiti
     copia:     Analisi
```

Il programma del Listato 6.3 permette invece di aggiungere a una variabile stringa i caratteri presenti in un'altra.

```
/* Concatenazione di due stringhe */

#include <stdio.h>

char frase[160] = "Analisi, requisiti ";
```

```
main()
{
char dimmi[80];
int i, j;

printf("Inserisci una parola: ");
scanf("%s", dimmi);
for(i=0; (frase[i])!='\0'; i++)
  ;
for(j=0; (frase[i]=dimmi[j])!='\0'; i++,j++)
  ;
printf("frase: %s \n", frase);
}
```

Listato 6.3 Concatenazione di array di caratteri

In questo caso indichiamo esplicitamente il numero di elementi (160) che compongono la variabile `frase`, poiché desideriamo definire un array che possa contenere più caratteri di quelli presenti nella stringa assegnatagli all'inizio (20):

```
char frase[160] = "Analisi, requisiti ";
```

In questo modo `frase` potrà contenere i caratteri che gli verranno concatenati. La prima istruzione `printf` richiede all'utente una stringa e `scanf` la inserisce nell'array di caratteri `dimmi`. In generale non si conosce il numero di caratteri che attualmente costituiscono la stringa di partenza, per cui si deve scorrerla fino a posizionare l'indice sul carattere terminatore:

```
for(i=0; (frase[i])!='\0'; i++)
       ;
```

Alla fine del ciclo `i` conterrà l'indice dell'elemento del vettore dov'è presente il carattere `\0`. Nel caso specifico, avendo assegnato a `frase` la stringa `"Analisi, requisiti "`, `i` avrà valore 20.

Adesso dobbiamo assegnare agli elementi successivi di `frase` il contenuto di `dimmi`:

```
for(j=0; (frase[i]=dimmi[j])!='\0'; i++,j++)
       ;
```

L'indice `j` scorre `dimmi` a partire dalla prima posizione, mentre `i` scorre `frase` a partire dal suo carattere terminatore; alla prima iterazione il carattere `\0` di `frase` viene sostituito dal primo carattere di `dimmi`. A ogni ciclo successivo viene assegnato a `frase[i]` il valore di `dimmi[j]`. All'ultima iterazione il carattere `\0` di `dimmi` viene posto in `frase`, così da chiuderla correttamente.

L'istruzione `printf` visualizza il nuovo contenuto di `frase`.

```
printf("frase: %s \n", frase);
```

Osserviamo di seguito una possibile esecuzione del programma.

```
Inserisci una parola: funzionali
     frase: Analisi, requisiti funzionali
```

In Figura 6.2, A e B corrispondono agli stati degli array immediatamente prima e dopo l'esecuzione del secondo `for` del programma che effettua la concatenazione.

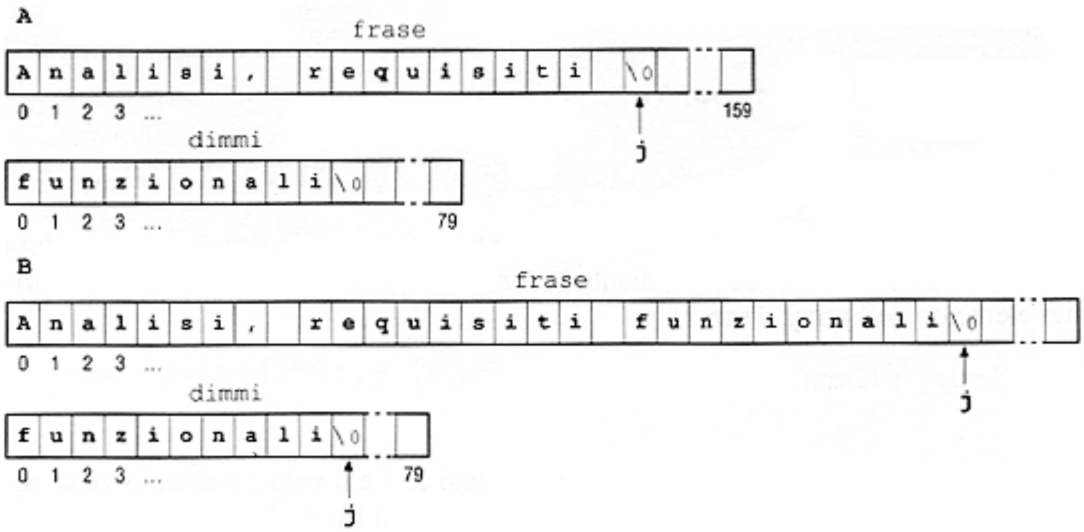


Figura 6.2 Stato degli array prima e dopo la concatenazione

Un altro modo per memorizzare una stringa è l'uso, all'interno di un ciclo, della funzione `getchar`, che cattura il carattere passatole in ingresso (Listato 6.4).

```
/* Concatenazione di due stringhe
   introduzione della seconda stringa con getchar */

#include <stdio.h>

char frase[160] = "Analisi, requisiti ";

main()
{
char dimmi[80];
int i, j;

printf("Inserisci una parola: ");
for(i=0; (dimmi[i]=getchar())!='\n'; i++)
  ;
dimmi[i]='\0';
for(i=0; frase[i]!='\0'; i++)
  ;
for(j=0; (frase[i]=dimmi[j])!='\0'; i++,j++)
  ;
printf(" frase: %s \n", frase);
}
```

Listato 6.4 Immissione di caratteri con `getchar()`

Il ciclo `for` che sostituisce l'istruzione `scanf` inizializza l'indice `i` a zero, cattura il carattere passato da tastiera mediante la funzione `getchar` e lo inserisce nel primo elemento dell'array `dimmi`:

```
for(i=0; (dimmi[i]=getchar())!='\n'; i++)
        ;
```

A ogni iterazione il valore di `i` viene incrementato di 1 e il valore immesso viene inserito nel corrispondente elemento dell'array. Il ciclo si ripete finché il carattere immesso è diverso da `\n`, cioè fino a quando l'utente non batte un Invio.

La stringa memorizzata in `dimmi` non contiene il carattere terminatore, che va esplicitamente assegnatogli nella posizione appropriata:

```
dimmi[i] = '\0';
```

È chiaro che potremmo decidere d'interrompere l'inserimento al verificarsi di un altro evento; per esempio, quando l'utente batte un punto esclamativo. In questo modo potremmo memorizzare più linee nello stesso array: ogni Invio dato dal terminale corrisponde infatti all'assegnamento di un `\n` a un elemento dell'array; evidentemente una successiva visualizzazione dell'array mostrerebbe la stringa con gli accapo inseriti dall'utente.

✓ **NOTA**

Nel programma abbiamo definito `dimmi` di 80 caratteri. Se l'utente ne inserisse un numero maggiore, come abbiamo già evidenziato, i sovrabbondanti andrebbero a sporcare zone contigue di memoria centrale. Il C non fa infatti nessun controllo automatico del rispetto dei margini dell'array. È il programmatore che si deve preoccupare di verificare che gli assegnamenti vengano effettuati su elementi definiti dell'array, per cui un più corretto ciclo d'inserimento del programma sarebbe:

```
for(i=0; ((dimmi[i]=getchar())!='\n') && (i<80)) ;i++)
  ;
```

Il ciclo prosegue finché il carattere catturato è diverso da `\n` e contemporaneamente `i` è minore di 80.

Scriviamo adesso un programma che confronta due stringhe rivelando se la prima è uguale, maggiore o minore della seconda (Listato 6.5). L'ordinamento seguito è quello definito dal codice di rappresentazione dei caratteri, che nella maggior parte delle macchine è il codice ASCII.

```
#include <stdio.h>

/* Confronto fra due stringhe */

char prima[160] = "mareggiata";

main()
{
char seconda[80];
int i;

printf("Inserisci una parola: ");
for(i=0; ((seconda[i]=getchar()) != '\n') && (i<80) ;i++)
  ;
seconda[i]='\0';
for(i=0; (prima[i] == seconda[i]) && (prima[i] != '\0') &&
    (seconda[i] != '\0'); i++)
  ;
if(prima[i]==seconda[i])
  printf("Sono uguali\n");
else
  if(prima[i]>seconda[i])
    printf("La prima è maggiore della seconda\n");
  else
    printf("La seconda è maggiore della prima\n");
}
```

Listato 6.5 Confronto fra array di caratteri

L'istruzione `for`

```
for(i=0; (prima[i]==seconda[i]) && (prima[i]!='\0') &&
```

```
        (seconda[i]!='\0'); i++);
```

scorre in parallelo gli elementi dei due array e li confronta; il ciclo si interrompe quando `prima[i]` non risulta essere uguale a `seconda[i]` oppure quando finisce una delle due stringhe. L'`if` seguente serve a determinare la ragione per cui il ciclo `for` si è interrotto; si noti che l'unica possibilità per cui `prima[i]` è uguale a `seconda[i]` si presenta quando entrambi sono uguali a `\0`, il che significa che le stringhe hanno la stessa lunghezza e sono uguali.

6.3 Funzioni di libreria

Esistono nella libreria `string.h` funzioni standard che permettono di effettuare le operazioni che abbiamo esaminato sulle stringhe e molte altre ancora. Come al solito, è sufficiente dichiarare il riferimento a tale libreria all'inizio del programma per poter utilizzare le funzioni in essa contenute.

La funzione `strcpy` consente di copiare `stringa2` su `stringa1`:

```
strcpy(stringa1, stringa2);
```

La funzione `strncpy` permette invece di copiare i primi *n* caratteri di `stringa2` in `stringa1`:

```
strncpy(stringa1, stringa2, n);
```

mentre la funzione `strcat` consente di concatenare `stringa2` a `stringa1`:

```
strcat(stringa1, stringa2);
```

La funzione `strcmp` serve a confrontare `stringa2` con `stringa1` (Listato 6.6):

```
strcmp(stringa1, stringa2);
```

Se risultano essere uguali viene restituito zero, se `stringa1` è maggiore di `stringa2` viene restituito un valore positivo, altrimenti un valore negativo ▬.

```
/* Confronto tra due stringhe con strcmp */

#include <stdio.h>
#include <string.h>

char prima[160] = "mareggiata";

main()
{
char seconda[80];
int i, x;

printf("Inserisci una parola: ");
for(i=0; ((seconda[i]=getchar())!='\n') && (i<80); i++)
  ;
seconda[i] = '\0';

if( (x = (strcmp(prima, seconda))) == 0)
  printf("Sono uguali\n");
else
  if(x>0)
    printf("la prima è maggiore della seconda\n");
  else
    printf("la seconda è maggiore della prima\n");
}
```

Listato 6.6 Esempio di utilizzo di strcmp

6.4 Esercizi

1. Scrivere un programma che, senza utilizzare la libreria `string.h`, concateni a una stringa i primi cinque caratteri di una seconda stringa.

2. Scrivere un programma che confronti due stringhe, limitatamente ai primi cinque caratteri, e successivamente visualizzi il risultato del confronto. Il programma non deve utilizzare la libreria `string.h`.

* 3. Scrivere un programma che ottenga lo stesso effetto richiesto nell'Esercizio 1 ma con l'utilizzo della funzione `strncat` della libreria `string.h`.

* 4. Scrivere un programma che ottenga lo stesso effetto richiesto nell'Esercizio 2 ma con l'utilizzo della funzione `strncmp` della libreria `string.h`.

5. Data la seguente assegnazione alla stringa `esercizio`

```
esercizio:='1234567890abcdefghilmopqrstvuzABCDEFGHILMNOPQRSTVUZ';
```

spostare i caratteri numerici dopo le lettere minuscole e prima delle lettere maiuscole, in modo che la stringa assuma il valore

```
abcdefghilmopqrstvuz1234567890ABCDEFGHILMNOPQRSTVUZ
```

Effettuare le operazioni necessarie senza utilizzare costanti che identifichino la posizione dei caratteri, ma reperire dinamicamente tali posizioni, in modo che il programma abbia una valenza più generale.

6. Scrivere un programma che, richieste all'utente le stringhe `frase`, `parola1` e `parola2`, controlli se in `frase` è contenuta `parola1`, e in tal caso sostituisca tutte le sue occorrenze con `parola2`.

7. Scrivere un programma che controlli se una stringa richiesta all'utente è palindroma. (Una stringa si dice palindroma se si legge nello stesso modo da sinistra verso destra e da destra verso sinistra. Sono esempi di stringhe palindrome: ANNA, radar, anilina.)

8. Scrivere un programma che richiesta all'utente una stringa controlli se vi compaiono almeno tre caratteri uguali consecutivi.

9. Scrivere un programma che richieda all'utente un carattere e una stringa e calcoli quindi il numero di occorrenze del carattere nella stringa.

10. Scrivere un programma che, letta una stringa composta da sole lettere dell'alfabeto, visualizzi il numero delle vocali, quello delle consonanti e la lettera più frequente.

11. Scrivere un programma che, letta una stringa composta da sole cifre (0..9), visualizzi accanto a ogni cifra il numero di volte che questa compare nella stringa. (Attenzione: si scriva un programma che utilizzi un solo ciclo.)

13. Scrivere un programma che richieda all'utente una stringa e ne visualizzi una seconda, ottenuta dalla prima sostituendo tutte le lettere minuscole con delle maiuscole.

7.1 Il concetto di sottoprogramma

Un programma è formato da elementi connessi in modo da raggiungere un determinato scopo. Le istruzioni possono essere considerate i componenti di un programma. Ciascuna istruzione corrisponde a un'azione elementare: ponendo le istruzioni in un determinato ordine il programma svolge il compito cui era stato destinato dal progettista. Da questo punto di vista, un programma non è diverso da un qualunque altro sistema; per esempio un personal computer è costituito da più elementi: la tastiera, il video, l'unità centrale, la stampante ecc., ognuno dei quali è connesso all'altro in un ordine specifico ed è preposto a uno specifico compito. A sua volta, ogni elemento potrebbe essere scomposto in ulteriori componenti. Se cominciassimo a smontare una stampante tra i pezzi componenti troveremmo una consolle di comando, il trattore della carta, la testina di stampa, il motore e così via. Se continuassimo a smontare la stampante (probabilmente distruggendola!) individueremmo centinaia di altri componenti prima di giungere ai pezzi non ulteriormente smontabili.

Un programma non può essere “smontato” oltre il limite delle singole istruzioni, ma è possibile aggregare gruppi di istruzioni per formare dei “semilavorati” detti *sottoprogrammi*. Come un personal computer è composto da tastiera, video, stampante e unità centrale, così un programma per il calcolo degli stipendi potrebbe essere scomposto nei sottoprogrammi di immissione delle ore lavorate, di calcolo dello stipendio e di visualizzazione e stampa della situazione contabile di ogni impiegato.
I sottoprogrammi si usano anche per evitare di replicare porzioni di codice sorgente: invocare un sottoprogramma significa mandare in esecuzione la porzione di codice corrispondente. Se un sottoprogramma è invocato più volte, la porzione di codice è eseguita più volte, tante quante sono le invocazioni. Il vantaggio dei sottoprogrammi è appunto di consentire al programmatore di avere tante chiamate ma una sola porzione di codice.
È possibile poi creare delle librerie, cioè delle raccolte di sottoprogrammi che possono essere utilizzati senza essere a conoscenza dei dettagli implementativi. È quanto avviene con le funzioni printf() e scanf(), la cui dichiarazione è contenuta nel file stdio.h.

7.2 Sottoprogrammi C

In C i sottoprogrammi sono detti *funzioni*: a partire da uno o più valori presi in ingresso, esse *ritornano* (o *restituiscono*) un valore al programma chiamante. Come indicato in Figura 7.1, una funzione può essere pensata come una *scatola nera* che a determinati valori in ingresso fa corrispondere un determinato valore in uscita.



Figura 7.1 La funzione come scatola nera

Un esempio di funzione C è abs(i), già utilizzata più volte. Considerando la funzione abs come una scatola nera, tutto quello che dobbiamo sapere – e in effetti già sappiamo – è che inserendo come argomento i di tale funzione un numero intero essa ne ritorna il valore assoluto (Figura 7.2).



Figura 7.2 La funzione abs()come scatola nera che produce il valore assoluto di i

Questo discorso è valido in generale: come programmatori possiamo utilizzare una qualsiasi funzione di libreria considerandola una scatola nera, cioè senza conoscere niente del suo funzionamento interno e interessandoci solo di cosa passarle in ingresso e di cosa viene restituito in uscita. Se però vogliamo creare noi stessi una nostra specifica funzione dobbiamo anche occuparci di come questa possa svolgere il compito affidatole.

Prima di esaminare la sintassi di dichiarazione e definizione di una funzione si consideri l’esempio riportato nel Listato 7.1.

```
#include <stdio.h>

double cubo(float);
```

```
main()
{
  float  a;
  double b;

  printf("Inserisci un numero: ");
  scanf("%f", &a);

  b = cubo(a);
  printf("%f elevato al cubo è uguale a %f", a, b);
}


double cubo(float c)
{
  return (c*c*c);
}
```

Listato 7.1 Dichiarazione, definizione e invocazione di una funzione

Per poter usare un identificatore occorre innanzitutto dichiararlo. La dichiarazione:

```
double cubo(float);
```

che precede main introduce l'identificatore cubo. Per mezzo di questa dichiarazione si specifica che cubo è il nome di una funzione che restituisce al programma chiamante un valore di tipo double. Inoltre si dichiara che la funzione cubo accetta in ingresso un solo valore come argomento, il cui tipo è float.
Si noti come con questa dichiarazione non si sia ancora definita la funzione cubo, cioè ancora non vengano specificate le istruzioni che caratterizzano la funzione; semplicemente abbiamo dichiarato un nuovo *nome*, cubo, e abbiamo detto a quale categoria appartiene questo nome. La definizione della funzione cubo avviene più tardi, dopo la fine del blocco di istruzioni di main:

```
double cubo(float c)
    {
       return (c*c*c);
    }
```

Oltre al nome della funzione viene definito il numero, il tipo e il nome dei suoi parametri, cioè le variabili su cui essa agisce. Nel nostro esempio è presente un solo parametro, il cui tipo è float e il cui nome è c. Il compito svolto da cubo è molto semplice: il valore passato nel parametro c è moltiplicato per se stesso tre volte (c*c*c) e il risultato di questa espressione è convertito in double e restituito (con return) al programma chiamante.
Il programma chiamante non ha da fare altro che passare alla funzione cubo un valore. Nell'esempio lo fa passando a cubo il valore contenuto nella variabile a: cubo(a). Successivamente il valore calcolato da cubo viene assegnato a una variabile di tipo double. Nell'esempio tale variabile è b e l'assegnazione è

```
b = cubo(a);
```

Un esempio di esecuzione è il seguente:

```
Inserisci il numero: 5
    5.000000 elevato al cubo è uguale a 125.000000
```

Con questo semplice esempio abbiamo messo in luce diversi aspetti della sintassi delle funzioni:

- • la dichiarazione di una funzione: double cubo(float);
- • la definizione di una funzione: double cubo(float c) {...};
- • il ritorno di un valore: return(c*c*c);
- • l'invocazione di funzione: b = cubo(a).

Passiamo ora a considerare in dettaglio ciascuno dei punti evidenziati.

7.3 Dichiarazione di una funzione

In termini generali una funzione viene dichiarata con la sintassi detta *prototyping*:

```
tipo_ritorno nome_funz (tipo_par1, ..., tipo_parN);
```

La dichiarazione introduce il nome della funzione, che in questo modo può essere utilizzato dal programma, ma presuppone che da qualche altra parte ne esista la definizione, altrimenti quel nome resterebbe privo di significato e il compilatore segnalerebbe un errore.
Il programmatore potrebbe anche scegliere di definire una funzione in un file e invocarla in un altro. In tal caso, nel file in cui viene utilizzata la funzione senza che essa vi sia definita l'invocazione della funzione deve essere preceduta da una sua dichiarazione. Per il momento consideriamo solo programmi interamente contenuti in un file; la dichiarazione di una funzione serve per poter separare il punto in cui una funzione è invocata dal punto in cui essa è definita. Avremo modo più avanti di approfondire l'argomento parlando di programmi su più file.
Nella dichiarazione di una funzione si potrebbero specificare anche i nomi dei parametri formali. Per esempio:

```
double cubo(float c);
```

è una dichiarazione valida. Il nome del parametro formale, però, è assolutamente superfluo. Ciò che conta in una dichiarazione è il tipo, o meglio la lista dei tipi dei parametri formali. Se in una dichiarazione di una funzione si specificano anche i nomi dei parametri formali il compilatore semplicemente li ignora.

7.4 Definizione di una funzione

In termini generali una funzione viene definita con la sintassi *prototyping* nel seguente modo:

```
tipo_ritorno nome_funz (tipo_par1 par1, ..., tipo_parN parN)
{
...
}
```

La definizione stabilisce il nome della funzione, i valori in ingresso su cui agisce – detti *parametri formali* –, il blocco di istruzioni che ne costituiscono il contenuto, e l'eventuale valore di ritorno. Per i nomi delle funzioni valgono le consuete regole in uso per gli identificatori. Nelle parentesi tonde che seguono il nome della funzione sono definiti i parametri formali specificandone il tipo e il nome.
Per ogni funzione introdotta nel programma occorre una definizione, ma si ricordi che in C non è ammesso che più funzioni abbiano lo stesso nome. Per esempio, le due definizioni, poste in uno stesso programma:

```
double cubo(float c)
{
   return (c*c*c);
}
```

```
float cubo(int c)
{
   return (c*c*c);
}
```

darebbero luogo a un errore pur avendo parametri diversi e ritornando valori di tipo diverso.
Nel blocco istruzioni delimitato da parentesi graffe può essere inserita qualunque istruzione, compresa una chiamata di funzione.
Studiamo ora il Listato 7.2.

```
#include <stdio.h>

double quad(float);
double cubo(float);
double quar(float);
double quin(float);
double pote(float, int);

main()
{
```

```
  int    base, esponente;
  double ptnz;

  printf(" Inserire base: " );
  scanf("%d", &base);
  printf(" Inserire esponente (0-5): ");
  scanf("%d", &esponente);

  ptnz = pote( base, esponente);

  if (ptnz == -1)
    printf("Potenza non prevista\n");
  else
    printf("La potenza %d di %d e' %f \n", esponente, base, ptnz);
}

double quad(float c)
{
  return(c*c);
}

double cubo(float c)
{
  return(c*c*c);
}

double quar(float c)
{
  return(c*c*c*c);
}

double quin(float c)
{
  return(c*c*c*c*c);
}

double pote(float b, int e)
{
  switch (e) {
    case   0: return (1);
    case   1: return (b);
    case   2: return (quad( b ));
    case   3: return (cubo( b ));
    case   4: return (quar( b ));
    case   5: return (quin( b ));
    default : return (-1);
  }
}
```

Listato 7.2 Dichiarazioni e definizioni di funzioni

La funzione `main` richiama la funzione potenza `pote` passando a essa due parametri attuali: `base` ed `esponente` (Figura 7.1).

```
ptnz = pote(base, esponente);
```

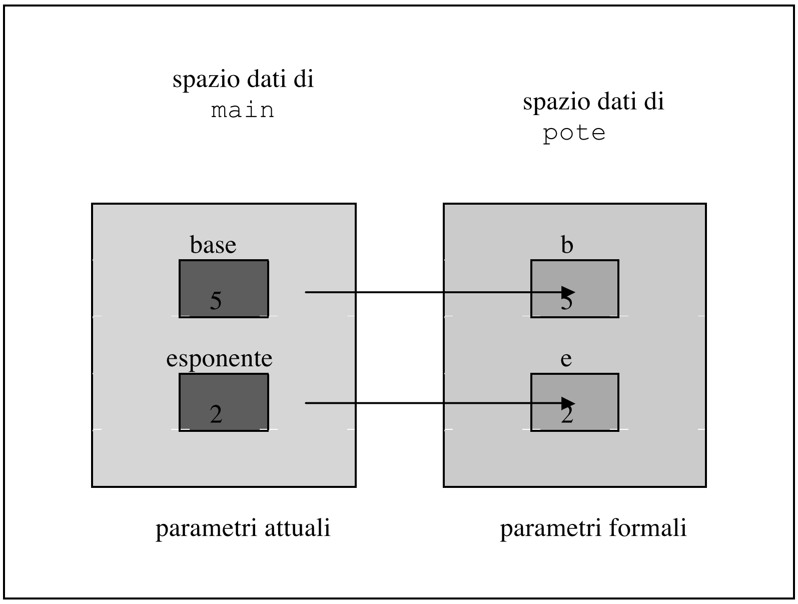


Figura 7.3 Passaggio di parametri tra `main` e `pote`

La funzione `pote`, che riceve in ingresso i valori nei parametri `b` ed `e` corrispondenti rispettivamente a `base` ed `esponente`, valuta il valore dell'esponente; dopo di ciò effettua una delle seguenti azioni: restituisce il valore 1 per esponente 0, la base stessa `b` per esponente 1, invoca la funzione `quad` per esponente 2, `cubo` per esponente 3, `quar` per esponente 4, `quin` per esponente 5 oppure restituisce –1 per segnalare la non disponibilità della potenza richiesta. Se l'esponente è 2, viene dunque invocata la funzione `quad`, cui la funzione `pote` trasmette il parametro attuale `b`:

```
case   2: return quad( b );
```

`quad` lo riceve in ingresso nel parametro formale `c` (Figura 7.2).

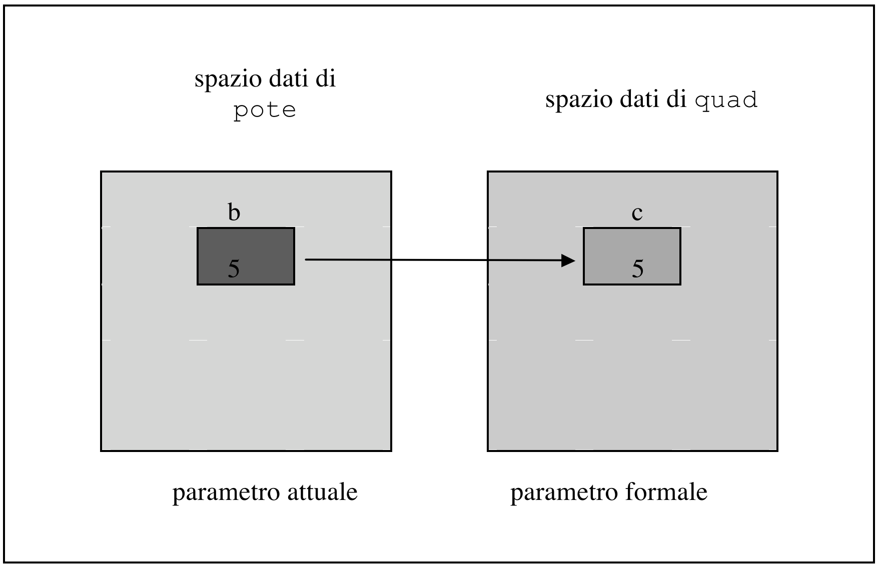


Figura 7.4 Passaggio di parametri tra `pote` e `quad`

La funzione `quad` calcola il quadrato di `c` e restituisce il risultato a `pote`, la quale a sua volta lo restituisce a `main` che l'aveva invocata (Figura 7.3). Il `main` gestisce tramite `if` il ritorno del valore negativo –1, usato per segnalare la non disponibilità della potenza richiesta.

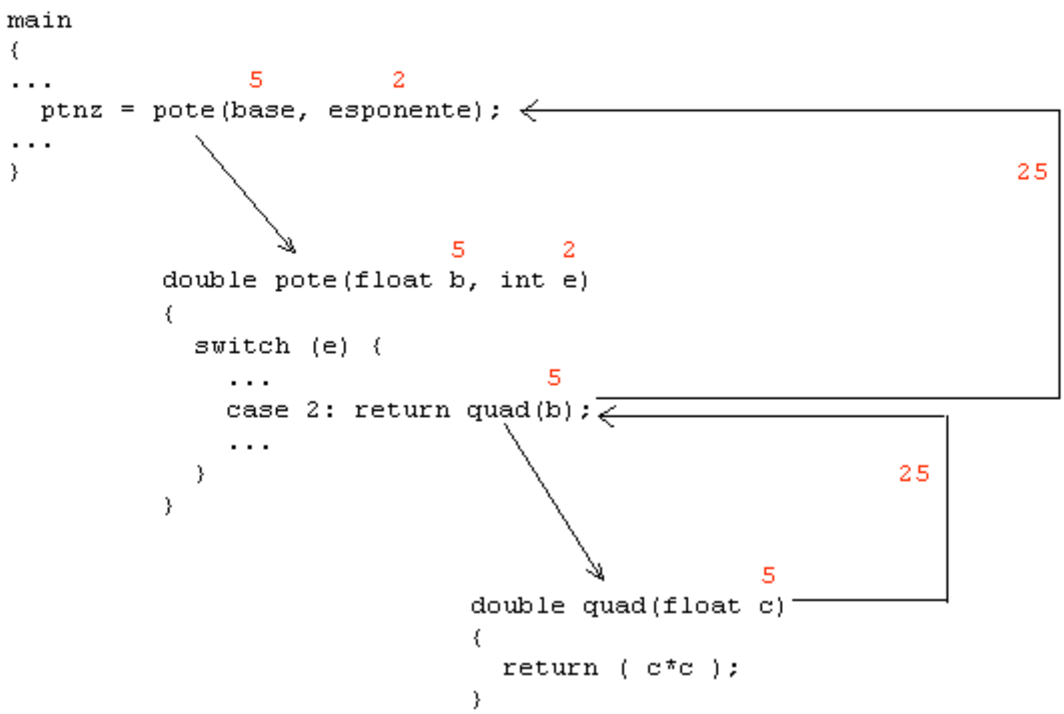


Figura 7.5 Esempio di chiamata e ritorno delle funzioni

✓ **NOTA**
In C, `main` non è una parola chiave del linguaggio ma il nome della funzione principale, cioè la funzione che, fra tutte quelle definite nel programma, viene eseguita per prima. La funzione `main` non è sintatticamente diversa dalle altre funzioni. La sua struttura:

```
main()
{
...
}
```

rispetta la sintassi generale di definizione delle funzioni. Alla funzione `main` possono essere anche passati dei parametri attraverso la linea di comando. Ritorneremo sull'argomento dopo aver trattato i puntatori ■.

7.5 Visibilità

Prima di passare all'elemento sintattico "`return`", dobbiamo osservare quanto segue.
Una dichiarazione introduce un nome in un determinato ambito di definizione, detto *scope*. In altre parole, ciò significa che un nome può essere usato soltanto – o, come si usa dire, è *visibile* – in una specifica parte del testo del programma.
Per un nome dichiarato all'interno del blocco istruzioni di una funzione (nome *locale*), la visibilità si estende dal punto di dichiarazione alla fine del blocco in cui esso è contenuto.
Per un nome definito al di fuori di una funzione (nome *globale*) la visibilità si estende dal punto di dichiarazione alla fine del file in cui è contenuta la dichiarazione. Così, per esempio, in

```
int x;

f()
{
   int y;

   y = 1;
}
```

la visibilità della variabile `y` si estende dal punto di definizione sino alla fine del blocco di appartenenza.
Anche i parametri formali di una funzione hanno un campo di visibilità che si estende dall'inizio alla fine del blocco istruzioni della funzione; sono quindi considerati a tutti gli effetti variabili locali alla funzione:

```
int x;

g(int y, char z)
{
   int k;
   int l;
   ...
}
```

Le variabili `y` e `z` sono locali alla funzione `g` e hanno una visibilità che si estende dalla parentesi graffa aperta `{` alla corrispondente parentesi graffa chiusa `}`. Quindi la definizione di `y` e `z` precede all'interno del blocco la definizione delle altre variabili locali `k` e `l` aventi anch'esse una visibilità che va dal punto di definizione alla fine del blocco ▬. Per questo motivo la funzione:

```
f(int x)
{
   int x;
}
```

è errata: in essa si tenta di definire due volte la variabile locale `x` nello stesso blocco.
Una dichiarazione di un nome in un blocco può nascondere, o come si dice in gergo, *mascherare*, la dichiarazione dello stesso nome in un blocco più esterno o la dichiarazione dello stesso nome globale. Un nome ridefinito all'interno di un blocco nasconde il significato precedente di quel nome, significato che verrà ripristinato all'uscita del blocco di appartenenza (Listato 7.3) ▬.

```
int x;              /* nome globale */

f()
{
  int x;            /* x locale che nasconde x globale */
  x = 1;            /* assegna 1 a x locale */
  {
    int x;          /* nasconde il primo x locale */
    x = 2;          /* assegna 2 al secondo x locale */
  }
  x = 3;            /* assegna 3 al primo x locale */
}

scanf ("%d", &x); /* inserisce un dato in x globale */
```

Listato 7.3 Esemplificazione di mascheramento dei nomi

Nell'esempio abbiamo tre diverse variabili `x`: la prima, definita al di fuori di qualunque blocco, è la variabile `x` globale, che in generale è visibile in tutto il file, ma viene mascherata dalla `x` locale, definita nel blocco più esterno della funzione `f()`. A sua volta questa `x` locale è nascosta dalla `x` del secondo blocco di `f()` interno al primo. All'uscita del blocco interno `x` denota il primo `x`, locale al blocco esterno, e all'uscita del blocco esterno `x` designa la variabile `x` globale.

✓ **NOTA**
È inevitabile che in un programma avvenga il mascheramento di nomi e non è infrequente il caso in cui il programmatore non si accorge di aver mascherato un nome all'interno di un blocco. È allora consigliato identificare le variabili globali con dei nomi caratteristici e univoci: usare per variabili globali nomi del tipo `i`, `j` oppure `x` significa rischiare mascheramenti indesiderati.

7.6 return

A ogni funzione C è associato un tipo, scelto tra quelli fondamentali o derivati, che caratterizza un valore. Questo valore è detto *valore di ritorno* della funzione ed è restituito dalla funzione al programma chiamante per mezzo dell'istruzione `return`. La sintassi da usare è:

```
return (espressione);
```

oppure:

```
return espressione;
```

Quando all'interno di un blocco di una funzione si incontra una istruzione `return`, il controllo viene restituito al programma chiamante, insieme a *`espressione`*. Per esempio, la funzione `cubo()`

```
double cubo( float c)
{
  return( c*c*c );
}
```

restituisce il controllo al programma chiamante e ritorna il cubo di `c` per mezzo dell'istruzione

```
return (c*c*c);
```

All'interno del blocco istruzioni di una funzione si possono avere più istruzioni `return`. Nella funzione `pote`:

```
double pote( b, e)
float b;
int   e;
{
  switch (e) {
    case   0: return 1;
    case   1: return b;
    case   2: return quad(b);
    case   3: return cubo(b);
    case   4: return quar(b);
    case   5: return quin(b);
    default : return -1;
  }
}
```

a ogni scelta del costrutto `switch-case` corrisponde un'uscita e la restituzione di un diverso valore. In questo caso abbiamo usato la forma sintattica del `return` non inserendo l'espressione di ritorno tra parentesi tonde.
L'invocazione di una funzione, detta anche *chiamata di funzione* (Paragrafo 7.8), può stare alla destra dell'operatore di assegnamento, salvo il caso in cui il tipo sia `void`. Nella funzione `cubo` del Listato 7.1, per esempio:

```
b = cubo(a);
```

alla variabile `b` viene assegnato il valore restituito dalla funzione `cubo`. Naturalmente il valore di ritorno può essere utilizzato all'interno di un'espressione:

```
y = a * cubo(x) + b * quad(x) + c * x + d;
```

Nel caso in cui non sia esplicitamente definito un tipo di ritorno il linguaggio C assume che esso sia il tipo fondamentale `int`.

7.7 Chiamata di una funzione

Una funzione C viene invocata facendo riferimento al nome e passando a essa una lista di parametri conforme in tipo, numero e ordine alla lista dei parametri formali elencata nella definizione della funzione stessa (Listato 7.4).

```
#include <stdio.h>

double area(float, float, char);

main()
{
  float b, h;
  double a;
  char p;

  printf("Inserire poligono (Triangolo/Rettangolo): ");
  scanf("%c", &p);

  printf("\nInserire base: ");
  scanf("%f", &b);
  printf("\nInserire altezza  : ");
  scanf("%f", &h);

  a = area( b, h, p);

  printf("Il poligono (b = %f, h = %f) ha area %f\n", b, h, a);
}

double area(float base, float altezza, char poligono)
{
  switch (poligono) {
    case 'T':     return (base * altezza/2.0);
    case 'R':     return (base * altezza);
    default :     return -1;
  }
}
```

Listato 7.4 Chiamata di funzione

Il contenuto delle variabili di tipo `float` `b`, `h` e di tipo `char` `p` vengono passati alla funzione `area` per mezzo dell'istruzione

```
a = area( b, h, p);
```

Le variabili `b`, `h` e `p` sono dette *parametri attuali* poiché contengono i valori di ingresso in quella specifica chiamata di funzione con i quali si può calcolare l'area del poligono. Al posto di una variabile si può comunque passare una costante dello stesso tipo, per esempio:

```
a = area( b, h, 'T');
```

dove il valore `T` viene immesso nel parametro formale `poligono`. Non sono invece corrette le invocazioni:

```
a = area('T', b, h);
a = area( b, h);
a = area( b, h, 'T', x);
a = area( b, h, x);
```

dove `x` è una variabile `int`; il passaggio di parametri è errato o per ordine o per numero o per discordanza di tipo.

✓ **NOTA**

Per mezzo della chiamata di funzione si concretizza il concetto di “scatola nera”: il programma chiamante sfrutta i servizi di una funzione conoscendo soltanto il nome e l’interfaccia (tipo, ordine e numero dei parametri formali) e disinteressandosi dei dettagli implementativi.
Le funzioni offrono anche un altro vantaggio: possono essere invocate quante volte lo si desidera senza produrre duplicazione di codice. In pratica, a n chiamate, con n ≥ 1, corrisponde sempre una sola definizione. Nel Listato 7.5, per esempio, la funzione `area` è chiamata due volte dalla funzione `main` per calcolare l’area del triangolo e del rettangolo, entrambi di base `b` e altezza `h` ▬.

```
#include <stdio.h>

double area(float, float, char);

main()
{
  float b, h;
  double tri, ret;

  printf("Inserire base: ");
  scanf("%f", &b);
  printf("Inserire altezza: ");
  scanf("%f", &h);

  tri = area(b, h, 'T');

  ret = area(b, h, 'R');

  printf("Il triangolo (b = %f, h = %f) ha area %f\n", b, h, tri);
  printf("Il rettangolo (b = %f, h = %f) ha area %f\n", b, h, ret);
}

double area(float base, float altezza, char poligono)
{
  switch (poligono) {
    case 'T':     return (base * altezza/2.0);
    case 'R':     return (base * altezza);
    default :     return -1;
  }
}
```

Listato 7.5 Le funzioni come strumento di riutilizzo del codice

7.8 Passaggio dei parametri

In questo capitolo abbiamo operato una distinzione tra due tipi di parametri: i *parametri formali* e i *parametri attuali*. I parametri formali sono quelli dichiarati per tipo, numero e ordine nella definizione della funzione. I parametri attuali sono invece quelli che vengono passati alla funzione all’atto della chiamata.
In C il passaggio dei parametri avviene sempre e soltanto *per valore*. Ciò significa che all’atto dell’invocazione di una funzione ogni parametro formale è inizializzato con il valore del corrispondente parametro attuale. Ecco perché deve esistere una coerenza di tipo e di numero tra parametri formali e parametri attuali. Occorre comunque chiarire che non è necessaria la *perfetta* corrispondenza. Infatti, nel trasferimento di valore da parametro attuale a parametro formale possono essere effettuate delle conversioni implicite di tipo. Per esempio, nel semplice programma:

```
main()
{
   double c;
```

```
   c = cubo( 2 );
}

double cubo(float c);
{
   return( c*c*c );
}
```

l'istruzione

```
c = cubo(2);
```

è perfettamente valida poiché la costante intera `2` viene convertita nella costante di tipo `double 2.0` (si ricordi che non esistono in C le costanti `float`).

Poiché con il passaggio dei parametri i valori dei parametri attuali sono travasati nelle locazioni di memoria corrispondenti ai parametri formali, si ha che la semantica del passaggio dei parametri è quella delle inizializzazioni di variabile: come per le inizializzazioni sono previste delle conversioni implicite di tipo. Più in dettaglio, si ha che nel passaggio dei parametri possono avvenire le conversioni seguenti.

`float`	I parametri attuali `float` sono convertiti in `double` prima di essere passati alla funzione. Di conseguenza tutti i parametri formali `float` sono automaticamente trasformati in `double`.
`char`	Tutti i parametri attuali `char` e `short int`, che esamineremo nei capitoli successivi, sono convertiti in `int`. Di conseguenza tutti i parametri formali `char` sono trasformati in `int`.

Occorre poi osservare che non è consentito il passaggio di parametri di tipo array, proprio perché in C il passaggio dei parametri avviene esclusivamente per valore. Infatti, se il compilatore si trovasse nella necessità di passare un array di tipo `int a[1000]`, occorrerebbe una quantità di tempo proporzionale per effettuare il travaso di valori tra due array di 1000 `int`.

Oltre al passaggio *esplicito* di parametri, è possibile anche il passaggio *implicito*. Infatti basta definire una variabile globale sia alla funzione chiamante sia a quella chiamata per ottenere la condivisione della variabile stessa. Si consideri l'esempio del Listato 7.6, in cui la variabile globale

```
char str[] = "Lupus in fabula";
```

è visibile sia dalla funzione `main` sia dalla funzione `lung_string`: quest'ultima fa riferimento a `str` per calcolarne il numero di caratteri, mentre la funzione `main` vi fa riferimento per visualizzarne il contenuto.

```
#include <stdio.h>

char str[] = "Lupus in fabula";

int lung_string(void);

main()
{
  int l;

  l = lung_string();
  printf("La stringa %s ha %d caratteri\n", str, l);
}

int lung_string(void)
{
  int i;
  for (i = 0; str[i] != '\0'; i++);
  return i;
}
```

Listato 7.6 Passaggio di parametri con variabile globale

✓ **NOTA**
Il passaggio implicito di parametri attraverso variabile globale è questione fortemente dibattuta. I dettami più severi della programmazione strutturata vorrebbero che i soli parametri passati a una funzione fossero quelli esplicitamente menzionati tra i parametri formali. Nella pratica non è tuttavia infrequente il caso di violazione di questa regola, soprattutto nelle applicazioni di tempo reale, in cui una variabile globale serve per il passaggio di dati tra due programmi (detti task) eseguiti in parallelo. Il lettore è comunque invitato a non abusare delle variabili globali: laddove è possibile è buona norma evitarle.

7.9 void

Abbiamo trattato il passaggio di parametri e la restituzione di un valore da parte di una funzione. Prendiamo ora in esame funzioni che non restituiscono alcun valore, e funzioni che non hanno parametri. In entrambi i casi il C mette a disposizione un "tipo" speciale detto `void`.
Tipico esempio di funzioni che non restituiscono alcun valore è quello delle funzioni il cui scopo è la visualizzazione di un messaggio o, più in generale, la produzione di un'uscita su uno dei dispositivi periferici. Queste funzioni sono talvolta conosciute con il curioso nome di funzioni "lavandino" (*sink*, in inglese) poiché prendono dati che riversano in una qualche uscita, senza ritornare niente al chiamante. Un esempio di funzione "lavandino" è la funzione `stampa_bin` (Listato 7.7).

```
#include <stdio.h>
#define DIM_INT 16

void stampa_bin ( int );

main()
{
  char resp[2];
  int  num;

  resp[0] = 's';

  while( resp[0] == 's' ) {
    printf("\nInserisci un intero positivo: ");
    scanf("%d", &num);
    printf("La sua rappresentazione binaria è: ");

    stampa_bin( num );

    printf("\nVuoi continuare? (s/n): ");
    scanf("%s",resp);
  }
}

void stampa_bin( int v )
{
  int i, j;
  char a[DIM_INT];

  if (v == 0)
    printf("%d", v);
  else  {
    for( i=0; v != 0; i++)    {
      a[i] = v % 2;
      v /= 2;
    }
```

```
    for(j = i-1 ; j >= 0; j--)
      printf("%d", a[j]);
  }
}
```

Listato 7.7 esempio di funzione “lavandino”

Un esempio di chiamata della funzione è:

```
Inserisci un intero positivo: 13
La sua rappresentazione binaria è: 1101
Vuoi continuare? (s/n): s
Inserisci un intero positivo: 64
La sua rappresentazione binaria è: 1000000
Vuoi continuare? (s/n): n
```

La funzione `stampa_bin` divide ripetutamente per 2 il decimale `v` e memorizza i resti delle divisioni intere nel vettore `a[]`, che poi legge a ritroso per visualizzare l’equivalente binario del decimale `v`. Come il lettore avrà osservato, sia nella dichiarazione

```
void stampa_bin( int );
```

sia nella definizione

```
void stampa_bin( int v )
{
...
}
```

si usa lo specificatore di tipo `void` per indicare l’assenza di un valore di ritorno. Viceversa, quando per una funzione non è specificato il tipo `void` per il valore di ritorno, nel blocco istruzioni della funzione è logico che sia presente per lo meno una istruzione `return`.
Il tipo `void` è usato anche per le funzioni che non assumono alcun parametro. Un esempio di funzione che non ha parametri e che non restituisce alcun valore è rappresentato da `mess_err` (Listato 7.8).

```
#include <stdio.h>

void mess_err( void );

main()
{
  int a, b, c;

  printf("Inserire dividendo:");
  scanf("%d", &a);
  printf("Inserire divisore:");
  scanf("%d", &b);
  if (b != 0) {
    c = a/b;
    printf("%d diviso %d = %d\n", a, b, c);
  }
else
  mess_err();
}

void mess_err( void )
{
  int i;
```

```
  char c;

  for (i = 0; i <= 20; i++) printf("\n");
  printf("                    ERRORE! DENOMINATORE NULLO");
  printf("\n Premere un tasto per continuare\n");
  scanf("%c%c", &c, &c);
}
```

Listato 7.8 Funzioni senza parametri

La funzione `mess_err` non prende parametri e non restituisce alcun valore; essa ha il solo compito di visualizzare un messaggio di errore nel caso di inserimento di un denominatore nullo. In C una funzione che non prende parametri può essere anche designata semplicemente dalle sole parentesi tonde, senza usare la parola chiave `void`. Per esempio:

```
void mess_err();
```

e

```
mess_err();
```

sono dichiarazioni valide. Nel secondo caso, però, `mess_err` viene considerata una funzione il cui eventuale valore di ritorno è di tipo `int`, anche se in realtà non ha nessun valore di ritorno. Non è forse questo il caso anche della funzione `main`? Abbiamo continuato a definirla con:

```
main()
{
...
}
```

a indicare il fatto che non ritorna nessun valore – è quindi di tipo `void` – e che non assume parametri. Una equivalente (e forse anche più corretta) definizione di `main` potrebbe essere:

```
void main( void )
{
...
}
```

Il fatto è che prima dello standard ANSI il C non prevedeva la parola chiave `void`, e oggi, per motivi di compatibilità, sono ammesse le due notazioni, con e senza `void`. Attualmente lo standard stabilisce che `main` sia implicitamente definita come funzione `void`, mentre in passato veniva comunemente definita di tipo `int`. Per evidenti motivi di leggibilità si consiglia caldamente di far uso di `void` tutte le volte che è necessario, soprattutto al fine di indicare che la funzione non ritorna nessun valore ▬.

7.10 La scomposizione funzionale

Vediamo ora alcuni criteri euristici per la progettazione di una funzione in C.
Il primo criterio riguarda la scelta del nome. Il nome di una funzione deve esprimere in sintesi la semantica, cioè qual è il compito della funzione. Saranno allora considerati opportuni nomi del tipo:

- `calcola_media`
- `acquisisci_valore`
- `converti_stringa`
- `tx_dati`
- `rx_dati`

mentre si sconsigliano nomi del tipo:

- `x_139       /* oscuro */`
- `gestore     /* generico */`
- `trota       /* umoristico ma poco significativo */`

La scelta del nome è anche un test della bontà della funzione. Se non si trova un nome che descrive sinteticamente il compito della funzione, probabilmente quest'ultima fa troppe cose, e quindi potrebbe essere ulteriormente scomposta, oppure non ha un compito preciso, nel qual caso potrebbe valere la pena eliminarla!
Una volta scelto il nome di una funzione si definiscono gli ingressi, che possono essere passati esplicitamente per valore o implicitamente per mezzo di variabili globali. In generale è sconsigliato, anche per problemi di leggibilità, avere una lista di parametri esageratamente nutrita. I parametri attuali e le variabili locali hanno vita temporanea, vengono creati all'atto dell'esecuzione della funzione e rimossi all'atto del ritorno al chiamante. Essi vengono posti dinamicamente in memoria; si ricordi che avere molti parametri significa avere molto consumo di memoria e di tempo di elaborazione.
La zona di memoria riservata alle chiamate di funzione viene gestita con la logica di una *pila* (*stack*), di cui parleremo diffusamente nel Capitolo 14. Cerchiamo comunque di rappresentare quello che accade in memoria durante il ciclo di vita di una funzione.
Quando viene invocata una funzione il sistema alloca uno spazio di memoria libero in testa alla pila, spazio riservato ai parametri formali e alle variabili locali. Per esempio, nel programma che calcola le potenze, dopo che il `main` ha chiamato la funzione `pote` la situazione è quella della Figura 7.4*a*. Se l'utente ha immesso il valore 2, `pote` chiama la funzione `quad` per determinare il quadrato del numero immesso e il sistema alloca spazio per i suoi dati (Figura 7.4*b*). Quando `quad` restituisce il controllo a `pote` la situazione ritorna a essere ancora quella di Figura 7.4*a*.

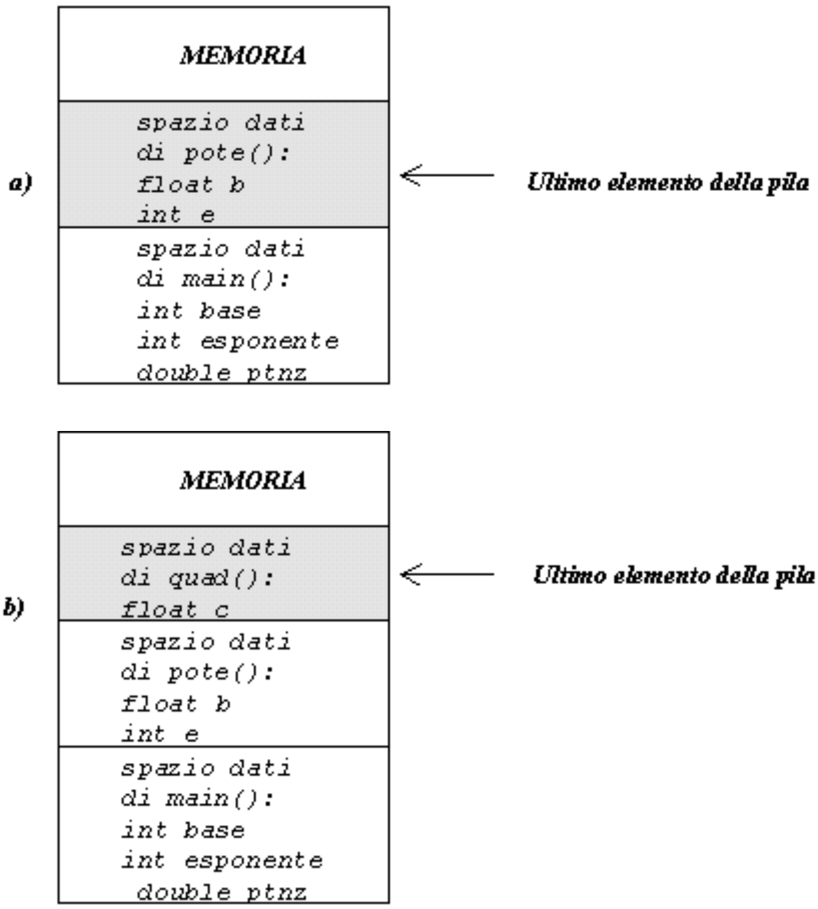


Figura 7.6 Allocazione dinamica di spazio di memoria sulla pila

Quindi, per esempio, definire una funzione come:

```
int f(void)
{
   char a[1000000];
   ...
}
```

potrebbe creare qualche problema! Ecco dunque una possibile eccezione alla sacre regole della programmazione strutturata: nei casi in cui una funzione debba lavorare su strutture di memoria costose in termini di occupazione è preferibile usare variabili globali.
L'ultima osservazione riguarda le uscite di una funzione. Abbiamo visto che in C una funzione ritorna al più un valore. Come ci si deve comportare allora quando è necessario che ci sia più di un valore di ritorno? Esistono due soluzioni:

1. 1. usare delle variabili globali;
2. 2. rendere note alla funzione delle locazioni in cui andare a depositare le uscite.

Per saperne di più su questo secondo tipo di soluzione, si rimanda al Capitolo 9 sui puntatori ▬.

7.11 Gestione di una sequenza

In questo paragrafo consideriamo il problema di far gestire all'utente una o più sequenze di interi mediante il seguente menu:

```
GESTIONE SEQUENZA


1. Immissione
2. Ordinamento
3. Ricerca completa
4. Ricerca binaria
5. Visualizzazione
0. fine

Scegliere una opzione:
```

Le opzioni possono essere scelte un numero di volte qualsiasi, finché non si seleziona la numero zero, che fa terminare il programma. Ovviamente, prima di tutto si deve scegliere la prima opzione per immettere la sequenza, ma successivamente questa possibilità può essere sfruttata per lavorare su altre sequenze.
Nel Listato 7.9 proponiamo il programma completo; dato che tutti gli algoritmi relativi sono stati visti nel Capitolo 5, adesso ci soffermiamo soltanto sull'uso delle funzioni e sul passaggio dei parametri.
L'array che conterrà la sequenza viene dichiarato come variabile globale:

```
int vet[MAX_ELE];  /* array che ospita la sequenza */
```

dunque tutte le funzioni del file vi possono accedere. (Nel Capitolo 9 vedremo una soluzione migliore.)
Decidiamo di far svolgere il compito di visualizzare il menu e gestire le scelte dell'utente alla funzione `gestione_sequenza`; essa dunque non dovrà restituire nessun valore e non accetterà nessun parametro:

```
void gestione_sequenza( void );
```

ma verrà semplicemente invocata dal `main`:

```
gestione_sequenza();
```

Viene naturale, poi, far corrispondere a ogni opzione una funzione che svolga il compito stabilito. Nel caso sia selezionata l'opzione 1 essa viene immessa nella variabile intera `scelta` e per mezzo del costrutto `switch-case` è mandata in esecuzione la funzione `immissione`:

```
case 1: n = immissione();
```

Essa deve ritornare a `gestione_sequenza` il numero di valori immessi in modo che esso possa essere reso noto alle altre funzioni; tale valore viene memorizzato nella variabile `n`. Se si verifica la dichiarazione di `immissione` si può vedere che effettivamente essa ritorna un intero.
Alla funzione `ordinamento` deve essere passato il numero di elementi della sequenza:

```
case 2: ordinamento( n );
```

Anch'essa agisce sulla variabile generale `vet[MAX_ELE]` ordinando i suoi elementi e non restituisce alcun valore: infatti il suo valore di ritorno è descritto in fase di dichiarazione e di definizione come `void`.

Nel caso di scelta 3 alla funzione `ricerca` deve essere passato, oltre alla lunghezza della sequenza, anche il valore dell'elemento da ricercare precedentemente richiesto all'utente in `gestione_sequenza`:

```
posizione = ricerca( n, ele );
```

La funzione ritorna un valore intero, che corrisponde alla posizione dove è stato reperito l'elemento. Considerazioni analoghe valgono per la funzione di ricerca binaria `ric_bin`.

```
#include <stdio.h>

#define MAX_ELE 1000       /* massimo numero di elementi */
int vet[MAX_ELE];          /* array che ospita la sequenza */

void gestione_sequenza( void );
int immissione( void );
void ordinamento( int );
int ricerca( int, int );
int ric_bin( int, int );
void visualizzazione( int );

main()
{
gestione_sequenza();
}

void gestione_sequenza()
{
int n;
int scelta = -1;
char invio;
int ele, posizione;

while(scelta != 0) {
 printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
 printf("\t\t\t GESTIONE SEQUENZA");
 printf("\n\n\n\t\t\t 1. Immissione");
 printf("\n\n\t\t\t 2. Ordinamento");
 printf("\n\n\t\t\t 3. Ricerca completa");
 printf("\n\n\t\t\t 4. Ricerca binaria");
 printf("\n\n\t\t\t 5. Visualizzazione");
 printf("\n\n\t\t\t 0. fine");
 printf("\n\n\n\t\t\t\t Scegliere una opzione: ");
 scanf("%d", &scelta);
 scanf("%c", &invio);
 printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

 switch(scelta) {
   case 1: n = immissione();
```

```
            break;
    case 2: ordinamento( n );
            break;
    case 3: printf("Elemento da ricercare: ");
            scanf("%d", &ele);
            scanf("%c", &invio);
            posizione = ricerca( n, ele );
            if(ele == vet[posizione])
              printf("\nElemento %d presente in posizione
                      %d\n", ele, posizione);
            else
              printf("\nElemento non presente!\n");
            printf("\n\n Premere Invio per continuare...");
            scanf("%c", &invio);
            break;
    case 4: printf("Elemento da ricercare: ");
            scanf("%d", &ele);
            scanf("%c", &invio);
            posizione = ric_bin( n, ele );
            if(posizione != -1)
              printf("\nElemento %d presente in posizione
                      %d\n", ele, posizione);
              else
              printf("\nElemento non presente!\n");
            printf("\n\n Premere Invio per continuare...");
            scanf("%c", &invio);
            break;
    case 5: visualizzazione( n );
            break;
  }
}
}

int immissione()
{
int i, n;
char invio;
do {
  printf("\nNumero elementi: ");
  scanf("%d", &n);
}
while (n < 1 || n > MAX_ELE);

for(i = 0; i < n; i++) {
  printf("\nImmettere un intero n.%d: ",i);
  scanf("%d", &vet[i]);
}
return( n );
}

void ordinamento( int n )
{
int i, p, k, n1;
int aux;
p = n; n1 = p;
do {
    k = 0;
    for(i = 0; i < n1-1; i++)
      if(vet[i] > vet[i+1]) {
```

```
        aux = vet[i]; vet[i] = vet[i+1];
        vet[i+1] = aux;
        k = 1; p = i + 1;
      }
    n1 = p;
}
while (k == 1);
}

/* Ricerca sequenziale */
int ricerca ( int n, int ele )
{
int i;
i = 0;
while (ele != vet[i] && i < n-1) ++i;
return( i );
}

/* ricerca binaria */
int ric_bin( int n, int ele )
{
int i, alto, basso, pos;
alto = 0; basso = n - 1; pos = -1;
do {
  i = (alto+basso)/2;
  if(vet[i] == ele) pos = i;
  else if(vet[i] < ele) alto = i + 1;
         else basso = i - 1;
}
while(alto <= basso && pos == -1);
return( pos );
}

void visualizzazione( int n )
{
int i;
char invio;
for(i = 0; i < n; i++)
  printf("\n%d", vet[i]);
printf("\n\n Premere Invio per continuare...");
scanf("%c", &invio);
}
```

Listato 7.9 Gestione di una sequenza con l'uso di funzioni per immissione, ordinamento, ricerca completa, ricerca binaria e visualizzazione

7.12 Esercizi

* 1. Scrivere una funzione che calcoli una qualunque potenza maggiore o uguale a zero.

* 2. Scrivere una funzione che visualizzi un messaggio a una generica coordinata *x*, *y* dello schermo. Si consideri uno schermo con 25 righe e 80 colonne.

* 3. Scrivere una funzione che, data una stringa di caratteri, converta tutte le lettere minuscole in maiuscole. Si passi la stringa tramite variabile globale.

* 4. Scrivere una funzione che calcoli il numero di caratteri e cifre presenti in una stringa passata come variabile globale.

* 5. Modificare la funzione di immissione della sequenza esaminata nell'ultimo paragrafo del presente capitolo in modo che dopo aver effettuato l'inizializzazione dell'array esso venga ordinato.

6. Modificare il programma del Listato 3.8, che calcola uno zero della funzione matematica $f(x) = 2x^3-4x+1$, in modo che utilizzi una funzione per determinare i valori di *f*.

7. Progettare e realizzare una funzione che accetti in ingresso una data e restituisca in uscita il corrispondente giorno della settimana. La funzione deve effettuare anche i controlli di validità della data immessa.

8. Progettare e realizzare una funzione che, data una stringa `s`, calcoli il numero di occorrenze del carattere `c` all'interno della stringa.

9. Modificare la funzione dell'Esercizio 1 esercizio in modo che calcoli anche le potenze negative.

10. Esaminare i programmi di questo capitolo e scrivere per ognuno di essi la lista delle variabili globali all'intero programma e di quelle locali a ogni sottoprogramma.

11. Scrivere una funzione che calcoli, al variare di *x*, il valore dell'espressione:

$$3x^3 - \sqrt{\frac{x^2+3}{2}}$$

12. Scrivere una funzione che visualizzi sullo schermo

```
                ROSSI & SARRI SpA
        Sistema per la gestione integrata

               OPZIONI DISPONIBILI

              1. Magazzino
              2. Clienti
              3. Fornitori
              4. Personale
              0. Fine

             Scegliere una opzione:
```

e quindi ritorni al programma chiamante la scelta effettuata dall'utente.

13. Scrivere una funzione che visualizzi la scritta:

```
Premere un tasto per continuare
```

e interrompa quindi l'esecuzione del programma chiamante finché non viene premuto un tasto.

14. Scrivere una procedura di conversione binario / decimale inversa a quella vista nel Listato 7.7. [*Suggerimento*: si consideri il numero binario come vettore di `char` di `0..1`.]

15. Modificare il programma di gestione sequenza (Listato 7.9) inserendo l'ulteriore opzione

5. inversione

Scrivere quindi il relativo sottoprogramma che visualizza la sequenza in ordine inverso.

8.1 Direttive

Un compilatore traduce le istruzioni di un programma sorgente in linguaggio macchina. Generalmente il programmatore non è consapevole del lavoro del compilatore: usa delle istruzioni in linguaggio di alto livello per evitare le idiosincrasie del linguaggio macchina. Talvolta però è conveniente prendere coscienza dell'esistenza del compilatore per impartirgli delle direttive.
In C è possibile inserire in un codice sorgente tali direttive, dette più propriamente *direttive del preprocessore*. Il preprocessore è il modulo del compilatore che scandisce per primo il codice sorgente e interpreta le direttive prima della traduzione in codice macchina. Le direttive del preprocessore non fanno realmente parte della grammatica del linguaggio, ma estendono l'ambiente di programmazione del C includendo il compilatore. Tra esse troviamo `define` e `include` , che abbiamo introdotto già dal Capitolo 1. Secondo lo Standard ANSI l'elenco delle direttive del preprocessore C è il seguente:

`#define`	`#error`	`#include`
`#elif`	`#if`	`#line`
`#else`	`#ifdef`	`#pragma`
#endif	`#ifndef`	`#undef`

Tutte iniziano con il simbolo # e una linea di codice non ne deve contenere più di una. La linea

```
#include "stdio.h" #include "stdlib.h"
```

produrrebbe perciò un messaggio di errore da parte del preprocessore.

8.2 #define

La direttiva `#define` è usata per associare una sequenza di caratteri a un identificatore. Il preprocessore, nel fare la scansione del codice sorgente, sostituisce a ogni occorrenza dell'identificatore la stringa di caratteri che vi è stata associata con la direttiva `#define`. Lo standard ANSI C si riferisce all'identificatore chiamandolo *nome-macro* e indica il processo di sostituzione con l'espressione *sostituzione di macro*.
La forma generale della direttiva è:

#define *nome-macro sequenza-caratteri*

Si osservi che questa – come tutte le altre direttive – non è terminata dal punto e virgola che chiude le istruzioni del linguaggio. Tra l'identificatore *`nome-macro`* e la sequenza di caratteri può esserci un qualsiasi numero di caratteri vuoti, ma una volta che la sequenza di caratteri ha inizio essa deve concludersi con un carattere di accapo. Per esempio,

se si vuole usare l'identificatore `VERO` per indicare il valore 1 e `FALSO` per il valore 0 si potrebbero dichiarare due macro:

```
#define VERO  1
#define FALSO 0
```

Questa direttiva fa sì che il preprocessore sostituisca a ogni occorrenza del nome `VERO` il numero `1` e a ogni occorrenza di `FALSO` il numero `0`. L'istruzione che segue verrebbe quindi visualizzata con la sequenza `0 1 2`:

```
printf("%d %d %d", FALSO, VERO, VERO+1);
```

Al compilatore questa istruzione appare come:

```
printf("%d %d %d", 0, 1, 1+1);
```

Una volta che è stato definito il nome di una macro, questo può essere riutilizzato come parte della definizione di altre macro. Per esempio, il frammento di codice che segue definisce i valori `UNO`, `DUE` e `TRE`:

```
#define UNO  1
#define DUE  UNO+UNO
#define TRE  UNO+DUE
```

La sostituzione di macro è un processo vantaggioso che risparmia al programmatore un lavoro ripetitivo e tedioso. Se per esempio si volesse definire un messaggio di errore standard si potrebbe usare una soluzione del tipo:

```
#define MYERR  "standard error on input\n"
...
printf(MYERR);
```

Il preprocessore procederebbe fedelmente a sostituire ogni occorrenza di `MYERR` con la stringa `"standard error on input\n"`. Il compilatore vedrebbe istruzioni del tipo:

```
printf("standard error on input\n");
```

e il programmatore avrebbe la garanzia di una messaggistica di errore uniforme, precisa e a basso costo.

✓ **NOTA**

Non si ha sostituzione di macro se il nome di macro è usato all'interno di una stringa:

```
#define XYZ  Buona Lettura
...
printf("XYZ");
```

L'ultima istruzione non visualizzerà la scritta `Buona Lettura` ma la stringa `XYZ`.

Nel caso in cui la sequenza di caratteri da associare a un *`nome-macro`* sia più lunga di una linea si può continuare su quella successiva terminando la linea con il carattere `\`. Per esempio:

```
#define STRINGONA "questa linea è molto molto molto \
molto molto molto lunga"
```

La possibilità della direttiva macro suggerisce alcune note di stile. Esiste una sorta di convenzione implicita per cui gli identificatori usati come *`nome-macro`* sono espressi in maiuscolo. Questa convenzione fa sì che a colpo d'occhio si possano identificare i punti in cui avrà luogo una sostituzione di macro. Secondariamente, prevale l'uso di concentrare le direttive `#define` in testa al file, o in un file di *include*, cioè un file che è oggetto di una direttiva `#include` (si veda il prossimo paragrafo), separato, piuttosto che disseminare il codice di direttive macro.

Il meccanismo delle macro viene tipicamente usato in C per associare un nome a una costante. Per esempio, se si è costruito un gruppo di funzioni che manipolano un array è conveniente rappresentare simbolicamente la dimensione dell'array invece di propagare nel codice il numero corrispondente a questo limite:

```
#define MAX_SIZE 100
	...
	float impiegato[MAX_SIZE];
```

Adottando questa soluzione si rende più stabile il codice. Infatti, per cambiare le dimensioni dell'array è sufficiente cambiare la corrispondente `#define`, in un solo punto, e ricompilare.

La direttiva `#define` possiede un'ulteriore caratteristica: una macro può avere argomenti. Una macro con argomenti è per molti aspetti simile a una funzione. Ogni qualvolta si incontra una macro nel codice gli argomenti dell'occorrenza sono sostituiti al posto di quelli della definizione:

```
#include "stdio.h"

	#define ABS(a)   (a)<0 ? -(a) : (a)

	main()
	{
	  printf("valore assoluto di -1 e 1: %d  %d", ABS(-1), ABS(1));
	  return 0;
	}
```

All'atto della compilazione l'identificatore `a` introdotto nella definizione della macro viene sostituito con i valori `-1` e `1`. Le parentesi tonde che circondano `a` garantiscono la correttezza della sostituzione. Potrebbero infatti verificarsi dei casi in cui la sostituzione provoca risultati non attesi. Per esempio:

```
ABS(10-20)
```

senza l'uso delle parentesi tonde verrebbe convertito in

```
10-20<0 ? -10-20 : 10 -20
```

producendo un risultato falso!

> ✓ **NOTA**
> La sostituzione di macro per la costruzione di funzioni presenta vantaggi e svantaggi. Il principale vantaggio è costituito dalle prestazioni. Infatti, producendo una espansione del codice localmente al punto in cui compare una occorrenza del `nome-macro` non si perde tempo nelle procedure di chiamata funzione. D'altra parte questa espansione locale della sequenza di caratteri duplica il codice e quindi aumenta le dimensioni del programma.
> Infine, occorre tenere presente che anche il programmatore C più esperto rimane talvolta vittima di sostituzioni non desiderate.

8.3 #include

La direttiva `#include`, di cui abbiamo sin qui fatto largo uso, dice al preprocessore di andare a leggere in un altro file sorgente oltre a quello che contiene la direttiva medesima. La sintassi generale è:

#include *nome-file*

Il qualificatore *`nome-file`* può essere indicato in due modi diversi: racchiuso da virgolette o da parentesi angolari:

```
#include "stdio.h"
#include <stdio.h>
```

Facendo precedere la parola chiave Se il nome del file è racchiuso tra parentesi angolari il preprocessore cercherà il corrispondente file secondo un percorso stabilito da chi ha realizzato il compilatore. La ricerca, insomma, avviene in qualche directory speciale appositamente creata all'atto dell'installazione del compilatore per includere i file. Se invece il nome è racchiuso tra virgolette il file è cercato in un altro modo, dipendente dall'implementazione. In

pratica ciò spesso vuol dire che la ricerca avviene nella directory di lavoro attuale. Se la ricerca del file dà esito negativo, il precompilatore la ripete come se il nome fosse stato racchiuso da parentesi angolari.

✓ **NOTA**
Conviene sempre consultare la guida del compilatore per conoscere con precisione come sono risolte parentesi angolari e virgolette.

Occorre ricordare che un file di *include* può a sua volta contenere altre direttive `#include`. Si parla in questo caso di *include* annidati. Il numero di livelli di annidamento varia da compilatore a compilatore; tuttavia lo standard ANSI stabilisce che siano disponibili per lo meno otto livelli di inclusione.

8.4 #error

Quando il compilatore incontra la direttiva `#error` visualizza un messaggio di errore. La sintassi di questa direttiva è:

#error *messaggio-errore*

Il termine `messaggio-errore` non è racchiuso tra doppi apici. La direttiva `#error` è usata per la correzione degli errori (*debugging*). Oltre al messaggio indicato dalla direttiva, il compilatore potrebbe aggiungere ulteriori informazioni sullo stato della compilazione.

8.5 Direttive condizionali di compilazione

Una comoda funzionalità offerta dal preprocessore C è quella delle compilazioni condizionali. Ciò significa che alcune porzioni di codice possono essere selettivamente compilate, per esempio, per includere o meno personalizzazioni dell'applicativo. Questa funzionalità è usata frequentemente quando si devono fornire diverse versioni di uno stesso programma su piattaforme differenti, per esempio Unix, Windows, Macintosh.

Le compilazioni condizionali si ottengono con le direttive `#if`, `#else`, `#elif` ed `#endif`, il cui significato è molto semplice. Se l'espressione costante che segue `#if` è vera, la porzione di codice compresa tra `#if` ed `#endif` viene compilata, altrimenti sarà ignorata dal compilatore. La direttiva `#endif` viene usata per marcare la fine del blocco `#if`. La sua forma sintattica generale è:

#if *espressione-costante*
    sequenza istruzioni
#endif

Per esempio il semplice programma seguente:

```
#include <stdio.h>

#define MAX 100

main()
{
  #if MAX>99
    printf("compilato per array maggiori di 99\n");
  #endif
}
```

mostrerebbe il messaggio argomento della `printf` poiché la costante simbolica `MAX` è maggiore di 99.

✓ **NOTA**
L'espressione che segue `#if`, se valutata, è valutata a tempo di compilazione. Ne consegue che essa deve contenere identificatori e costanti precedentemente definiti, ma non variabili. Il contenuto di una variabile può essere noto solo a tempo di esecuzione.

La direttiva `#else` ha lo stesso significato di `else` nelle istruzioni condizionali: stabilisce un'alternativa nel caso in cui `#if` sia valutato falso:

```
#include <stdio.h>

#define MAX 10

main()
{
  #if  MAX>99
    printf("compilato per array maggiori di 99\n");
  #else
    printf("compilato per array piccoli\n");
  #endif
}
```

In questo caso `MAX` è inferiore a 99, motivo per cui la porzione di codice che segue `#if` non è compilata, mentre lo è il codice che segue `#else`. In altre parole, sarà mostrato il messaggio `compilato per array piccoli`.
Si osservi come il termine `#else` venga usato sia per marcare la fine del blocco `#if` sia per segnare l'inizio del blocco `#else`. Ciò è necessario perché in una direttiva `#if` può essere presente un solo termine `#endif`.
La direttiva `#elif` significa "`else if`" ed è usata per stabilire una catena di "`if-else-if`" che realizza una compilazione condizionale multipla. Se l'espressione è vera, la sequenza di istruzioni è compilata e nessun altra sequenza `#elif` è valutata; altrimenti si controlla la prossima serie. La forma generale è:

```
#if espressione
    sequenza istruzioni
  #elif espressione 1
    sequenza istruzioni
  #elif espressione 2
    sequenza istruzione
  ...
  #elif espressione N
    sequenza istruzioni
  #endif
```

Per esempio, il seguente frammento usa il valore `COUNTRY` per definire la moneta corrente:

```
#define US 0
  #define ENGLAND 1
  #define ITALY 2

  #define COUNTRY ITALY

  #if COUNTRY==ITALY
    char moneta[]="lit";
  #elif CONTRY==US
    char moneta[]="dollar";
  #else
    char moneta[]="pound";
  #endif
```

Le direttive `#if` ed `#elif` possono essere annidate per un numero di livelli dipendente dal particolare compilatore. Le direttive `#endif`, `#else` ed `#elif` si associano con la direttiva `#if` o `#elif` più prossima. Per esempio:

```
#if MAX>100
    #if SERIAL_VERSION
      int port=198;
    #elif
      int port=200;
    #endif
  #else
```

```
  char out_buffer[100];
#endif
```

Nell'espressione di `#if` o di `#elif` si può usare l'operatore di preprocessore `defined` per determinare l'esistenza di una macro. Nella forma più generale si ha:

#if defined *nome-macro*
    sequenza istruzioni
  #endif

Se *`nome-macro`* è stato definito, la sequenza di istruzioni viene regolarmente compilata, altrimenti è ignorata. Si veda per esempio:

```
#include <stdio.h>

#define DEBUG

main()
{
  int i = 100;

  #if defined DEBUG
    printf("la variabile i vale: %d\n", i);
  #endif
}
```

Facendo precedere la parola chiave `define` dal simbolo `!` si ottiene una compilazione condizionale nel caso in cui la macro non sia stata definita.
Altre due direttive per la compilazione condizionale sono `#ifdef`, che significa "`if defined`", e `#ifndef`, che significa "`if not defined`". La sintassi generale è:

#ifdef *nome-macro*
    sequenza istruzioni
  #endif

Se il *`nome-macro`* è stato precedentemente definito da una `#define`, allora la sequenza di istruzioni verrà compilata.

✓ **NOTA**
Usare `#ifdef` è equivalente a usare `#if` insieme all'operatore `defined.`

La sintassi di `#ifndef` è:

#ifndef *nome-macro*
    sequenza istruzioni
  #endif

Se il *`nome-macro`* non è stato definito da alcuna `#define`, allora la sequenza istruzioni verrà compilata. Sia `#ifdef` sia `#ifndef` possono far uso della clausola `else`, ma non di quella `#elif`. Per esempio, il programma:

```
#include <stdio.h>

#define UGO 10

main()
{
  #ifdef UGO
    printf("Ciao Ugo\n");
  #else
    printf("Ciao a tutti\n");
  #endif
```

```
  #ifndef ADA
    printf("Ada non definita\n");
  #endif
}
```

una volta eseguito visualizzerà:

```
Ciao Ugo
Ada non definita
```

Anche `#ifdef` e `#ifndef` possono essere annidati secondo le solite regole della direttiva `#if`.

8.6 #undef

La direttiva `#undef` rimuove un nome di macro definito da una precedente `#define`. La sintassi è:

#undef *nome-macro*

Vediamo un esempio:

```
#define LEN 100
#define WIDTH 100

char array[LEN][WIDTH];

#undef LEN
#undef WIDTH
/* da questo punto in poi le costanti simboliche
   LEN e WIDTH non sono più definite          */
```

La direttiva `#undef` è usata per confinare la definizione di macro in precise porzioni di codice.

8.7 #line

La direttiva `#line` è usata per modificare il contenuto di due identificatori predefiniti dal compilatore i cui nomi sono `__LINE__` e `__FILE__`. La sintassi di questa direttiva è:

#line *numero* "*nomefile*"

dove `numero` è un qualsiasi intero positivo e *`nomefile`* è un qualunque identificatore valido di file. Il *`numero`* diviene il numero della linea di codice attuale del sorgente, e il nome del file diviene il nome del file sorgente. Il nome del file è opzionale.

La direttiva `#line` viene usata prevalentemente per il debug di applicazioni. L'identificatore `__LINE__` è un intero, mentre `__FILE__` è una stringa. Vediamo ora un esempio:

```
#include <stdio.h>

#line 100 /* inizializza il contatore di linea */
main()                /* linea 100 */
{                     /*linea 101 */
  printf("%d\n",__LINE__);  /*linea 102*/
}
```

In questo programma la direttiva `#line` stabilisce che il contatore di linee inizierà a 100. Come conseguenza la `printf` stamperà il valore di linea attuale, pari a 102.

8.8 #pragma

La direttiva #pragma è una direttiva definita dall'implementazione del particolare compilatore per istruire quest'ultimo. Per esempio, con la direttiva #pragma tipicamente si istruisce il compilatore a effettuare l'esecuzione passo passo del programma. Per maggiori dettagli si consiglia di consultare il manuale del compilatore.

8.9 Gli operatori # e

Lo standard ANSI C fornisce due operatori di preprocessore indicati dai simboli # e ##. Questi operatori sono usati all'interno delle macro.
All'interno di una #define applicare l'operatore # significa trasformare il suo argomento in una stringa delimitata da doppi apici:

```
#include <stdio.h>

    #define makestring(s) # s

    main()
    {
      printf(makestring(Mi piace il C));
    }
```

Il preprocessore trasforma l'istruzione

```
printf(makestring(Mi piace il C));
```

in

```
printf("Mi piace il C");
```

L'operatore ## è invece usato per concatenare due sequenze di caratteri in una #define:

```
#include <stdio.h>

    #define concat(a, b) a##b

    main()
    {
      int xy = 10;
      printf("%d", concat(x, y));
    }
```

Il preprocessore trasforma la curiosa istruzione

```
printf("%d", concat(x, y));
```

in

```
printf("%d", xy);
```

✓ **NOTA**

Se questi operatori sembrano strani non c'è da preoccuparsi. In effetti sono raramente usati nei programmi C e servono più che altro al preprocessore per gestire alcuni casi speciali.

8.10 Macro predefinite

Lo standard ANSI C specifica cinque macro predefinite:

__LINE__
__FILE__
__DATE__
__TIME__
__STDC__

Un compilatore standard deve prevedere queste macro. In generale, poi, un compilatore introduce numerose altre macro, specifiche di quella particolare implementazione.

Il significato di `__LINE__` e `__FILE__` è evidente. La prima corrisponde alla riga corrente e la seconda al nome del file corrente.

La macro `__DATE__` è una stringa costruita secondo il formato *mese/giorno/anno* che riporta la data dell'ultima compilazione del codice sorgente.

L'ora dell'ultima compilazione invece è conservata in `__TIME__` , stringa che assume la forma *ora:minuti:secondi*.

Infine, la macro `__STDC__` contiene la costante decimale 1. Ciò significa che l'implementazione del compilatore è conforme allo standard. Se invece riporta un qualsiasi altro numero l'implementazione non è standard.

8.11 Esercizi

1. 1. Realizzare la macro `ABS(X)` che calcola il valore assoluto di `X`.
2. 2. Realizzare la macro `CUBO(X)` che verrà espansa al valore di `X` elevato alla terza potenza. Quali valutazioni possiamo fare, sia in termini di efficienza sia più in generale, scegliendo di calcolare il valore assoluto e il cubo di un numero con una funzione o con una macro?
3. 3. Date le macroistruzioni

```
#define DIM 100;
#define VERO (a>100);
```

quale errore abbiamo probabilmente commesso nello scrivere le macroistruzioni (che comunque verranno accettate ed espanse dal precompilatore)? Come verrebbero espanse le seguenti istruzioni?

```
1. n = DIM;
2. float array[DIM];
3. while VERO
     Calcola();
```

In quale caso si avrà un errore in fase di compilazione? In quale caso un effetto indesiderato? E in quale tutto andrà bene ma avrà luogo un effetto ininfluente ma probabilmente non previsto?

4. 4. Supponiamo che `PIPPO` sia una macro già definita:

```
#if PIPPO == 100
  #undef PIPPO
  #define PIPPO 0
#if PIPPO == 200
  #undef PIPPO
  #define PIPPO 300
#else
  #define PLUTO 1000
#endif
```

Quali azioni vengono intraprese in dipendenza del valore iniziale di `PIPPO`? Quale differenze possiamo notare tra le istruzioni condizionali del preprocessore e le istruzioni condizionali del linguaggio C?

9.1 Definizione di puntatore

A ogni variabile corrisponde un nome, una locazione di memoria, e l'indirizzo della locazione di memoria. Il nome di una variabile è il simbolo attraverso cui si fa riferimento al contenuto della corrispondente locazione di memoria. Così, per esempio, nel frammento di programma:

```
int a;
     ...
     a = 5;
     printf("%d", a);
```

viene assegnato il valore costante 5 alla variabile di tipo intero `a`.



L'operatore `&`, introdotto con la funzione `scanf`, restituisce l'indirizzo di memoria di una variabile. Per esempio l'espressione `&a` è un'espressione il cui valore è l'indirizzo della variabile `a`. Un indirizzo può essere assegnato solo a una speciale categoria di variabili dette *puntatori*, le quali sono appunto variabili abilitate a contenere un indirizzo. La sintassi di definizione è

```
tipo_base *var_punt;
```

dove `var_punt` è definita come variabile di tipo "puntatore a *`tipo_base`*"; in sostanza *`var_punt`* è creata per poter mantenere l'indirizzo di variabili di tipo *`tipo_base`*, che è uno dei tipi fondamentali già introdotti: `char`, `int`, `float` e `double`.
Il tipo puntatore è un classico esempio di tipo derivato; infatti, non ha senso parlare di tipo puntatore in generale, ma occorre sempre specificare a quale tipo esso punta. Per esempio, in questo caso:

```
int   a;
char  c;

int  *pi;
char *pc;

pi = &a;
pc = &c;
```

si ha che `pi` è una variabile di tipo puntatore a `int`, e `pc` è una variabile di tipo puntatore a `char`. Le variabili `pi` e `pc` sono inizializzate rispettivamente con l'indirizzo di `a` e di `c`.



La capacità di controllo di una variabile o, meglio, la capacità di controllo di una qualsiasi regione di memoria per mezzo di puntatori è una delle caratteristiche salienti del C. Il lettore avrà modo di sperimentare quanto si accrescano le capacità del linguaggio con l'introduzione dei puntatori.
Questi ultimi, d'altra parte, sarebbero poca cosa se non esistesse l'operatore unario `*`. L'operatore `*`, detto *operatore di indirezione*, si applica a una variabile di tipo puntatore e restituisce il contenuto dell'oggetto puntato. Se effettuiamo le operazioni

```
a = 5;
c = 'x';
```

in memoria abbiamo la situazione illustrata di seguito.



Le istruzioni

```
printf("a = %d    c = %c", a, c);
printf("a = %d    c = %c", *pa, *pc);
```

hanno esattamente lo stesso effetto, quello di visualizzare:

```
a = 5    c = x
```

Vediamo un altro esempio:

```
char  c1, c2;
char  *pc;
...
c1 = 'a';
c2 = 'b';
printf(" c1 = %c, c2 = %c \n", c1, c2);

pc = &c1;          /* pc contiene l'indirizzo di c1 */
c2 = *pc;          /* c2 contiene il carattere 'a'  */
printf(" c1 = %c, c2 = %c \n", c1, c2);
```

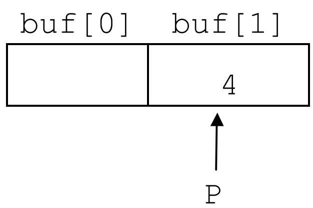
Dopo l'assegnazione `pc=&c1;` i nomi `c1` e `*pc` sono perfettamente equivalenti (*alias*), e si può accedere allo stesso oggetto creato con la definizione `char c1` sia con il nome `c` sia con l'espressione `*pc`. L'effetto ottenuto con l'assegnazione `c2=*pc` si sarebbe ottenuto, equivalentemente, con l'assegnazione

```
c2 = c1;
```

Un ulteriore esempio di uso di puntatori e dell'operatore di indirezione, riferiti a elementi di un array, è il seguente:

```
int  buf[2];
int  *p;
...
p  = &buf[1];
*p = 4;
```

Con il puntatore a intero `p` e l'operatore `*` si è modificato il contenuto della locazione di memoria `buf[1]`, questa volta preposta a contenere un valore di tipo `int`.



Il lettore avrà certo notato che l'operatore `*` è usato nella definizione di variabili di tipo "puntatore a":

```
int  *pi;
char *pc;
```

La notazione è perfettamente coerente con la semantica dell'operatore di indirezione. Infatti, se `*pi` e `*pc` occupano tanta memoria quanto rispettivamente un `int` e un `char`, allora `pi` e `pc` saranno dei puntatori a `int` e a `char`.

9.2 Array e puntatori

Gli array e i puntatori in C sono strettamente correlati. Il nome di un array può essere usato come un puntatore al suo primo elemento. Considerando, per esempio:

```
char  buf[100];
char  *s;

s = &buf[0];
s = buf;
```

si ha che le due assegnazioni `s=&buf[0]` e `s=buf` sono perfettamente equivalenti. Infatti in C il nome di un array, come nel nostro caso `buf`, è una costante – si noti bene: una costante, non una variabile! – il cui valore è l'indirizzo del primo elemento dell'array. Allora, come gli elementi di un array vengono scanditi per mezzo dell'indice, equivalentemente si può avere accesso agli stessi elementi per mezzo di un puntatore. Per esempio consideriamo il seguente codice:

```
char  buf[100];
char  *s;
...
s = buf;
buf[7] = 'a';
printf("buf[7] = %c\n", buf[7]);

*(s + 7)= 'b';
printf("buf[7] = %c\n", buf[7]);
```

Si ha che `s` e `buf` sono due sinonimi, con la differenza che `s` è una variabile puntatore a carattere, mentre `buf` è una costante. Incrementato di 7 il valore di `s` si ottiene un valore corrispondente all'indirizzo dell'ottavo elemento, al quale si accede per mezzo dell'operatore di indirezione. Il valore del puntatore `s` è incrementato di 7 unità, cioè di 7 volte la dimensione dell'oggetto corrispondente al tipo base del puntatore (Figura 9.1).

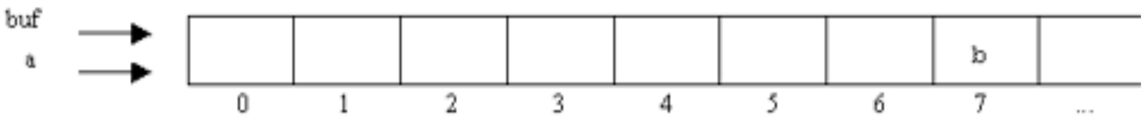


Figura 9.1 Si può far riferimento a `buf[7]` scrivendo `*(s+7)`

Per le stesse ragioni avremmo potuto riscrivere il frammento di programma

```
char buf[2];

for (i = 0; i < 2; i++)
   buf[i] = 'K';
```

per l'inizializzazione degli elementi di un array come

```
char *s;
char buf[2];

s = buf;
for (i = 0; i < 2; i++)
   *s++ = 'K';
```

L'istruzione `*s++='K'` opera nel modo seguente:

- copia `K` nella locazione di memoria puntata da `s`
- poi incrementa `s` di un elemento.

✓ **NOTA**
Se per errore avessimo scritto:

```
char *s;
char buf[2];

s = buf;
for (i = 0; i < 100; i++) *s++ = 'K';
```

il compilatore non avrebbe segnalato alcun errore, ma avremmo avuto problemi in esecuzione perché si sarebbe inizializzata con `'K'` una regione di memoria al di là del limite allocato con `buf`. Che si usi una variabile puntatore o si faccia riferimento alla notazione con indice, è sempre dovere del programmatore assicurarsi che le dimensioni di un array vengano rispettate.

9.3 Aritmetica dei puntatori

Un puntatore contiene un indirizzo e le operazioni che possono essere compiute su un puntatore sono perciò quelle che hanno senso per un indirizzo. Le uniche operazioni ammissibili sono dunque: l'incremento, per andare da un indirizzo più basso a uno più alto, e il decremento, per andare da un indirizzo più alto a uno più basso. Gli indirizzi sono per la memoria quello che sono gli indici per un array.
Gli operatori ammessi per una variabile di tipo puntatore sono:

```
+   ++   -   --
```

Ma qual è l'esatto significato dell'incremento o decremento di un puntatore? Non pensi il lettore che il valore numerico del puntatore corrispondente a un indirizzo venga incrementato come una qualunque altra costante numerica. Per esempio, se `pc` vale 10, dove `pc` è stato dichiarato:

```
char *pc;
```

non è detto che `pc++` valga 11 !
Nell'aritmetica dei puntatori quello che conta è il tipo base. Incrementare di 1 un puntatore significa far saltare il puntatore alla prossima locazione corrispondente a un elemento di memoria il cui tipo coincide con quello base. Per esempio, in:

```
int a[10];
char b[10];
int *pi;
char *pc;

pi = a;
pc = b;

pi = pi + 3;
pc = pc + 3;
```

le ultime due istruzioni che incrementano di 3 i puntatori `pi` e `pc` debbono essere interpretate in modo diverso. La prima,

```
pi = pi + 3;
```

significa spostare in avanti `pi` di tre posizioni, dove ogni posizione occupa lo spazio di un `int`. La seconda,

```
pc = pc + 3;
```

significa spostare in avanti `pc` di tre posizioni, dove ogni posizione occupa lo spazio di un `char`.
Più in generale si ha che, quando un operatore aritmetico è applicato a un puntatore `p` di un certo tipo e `p` punta a un elemento di un array di oggetti di quel tipo, `p+1` significa "prossimo elemento del vettore" mentre `p-1` significa "elemento precedente".
La sottrazione tra puntatori è definita solamente quando entrambi i puntatori puntano a elementi dello stesso array. La sottrazione di un puntatore da un altro produce un numero intero corrispondente al numero di posizioni tra i due elementi dell'array. Si osservi invece come sommando o sottraendo un intero da un puntatore si ottenga ancora un puntatore. Si considerino i tre esempi seguenti.

```
int v1[10];
int v2[10];
int i;
int *p;


i = &v1[5] - &v1[3];   /* 1 ESEMPIO */
printf("%d\n", i);     /* i vale 2 */

i = &v1[5] - &v2[3];   /* 2 ESEMPIO */
printf("%d\n", i);     /* il risultato è indefinito */

                       /* 3 ESEMPIO */
p = v2 - 2;            /* dove va a puntare p ? */
```

1. 1. *sottrazione tra indirizzi dello stesso vettore:*

   ```
   i = &v1[5] - &v1[3];
   ```

   corrispondente a un caso perfettamente legale;

2. 2. *sottrazione tra indirizzi di array diversi:*

   ```
   i = &v1[5] - &v2[3];
   ```

   corrispondente a un caso il cui risultato non è prevedibile;

3. 3. *sottrazione di una costante da un indirizzo ma nella direzione sbagliata:*

   ```
   p = v2 - 2;
   ```

   il puntatore `p` va a puntare due interi prima dell'inizio del vettore `v2` (per come sono avvenute le definizioni probabilmente si sconfina nello spazio riservato a `v1`, ma non è detto !).

9.4 Passaggio di parametri per indirizzo

Abbiamo osservato nel precedente capitolo che in C non è possibile passare un array a una funzione. Eppure esistono molti casi in cui è necessario non solo passare un array ma anche restituire una struttura dati più complessa della semplice variabile `char` o `int`.
All'apparenza le funzioni sembrano essere limitate dal meccanismo del passaggio parametri per valore. Il programmatore C risolve questa apparente pecca con un metodo semplice: passa per valore l'indirizzo della variabile – array o altro – che si vuol leggere o modificare tramite la funzione. Passare un indirizzo a una funzione significa renderle nota la locazione dell'oggetto corrispondente all'indirizzo. In tale maniera le istruzioni all'interno di una funzione possono modificare il contenuto della variabile il cui indirizzo è stato passato alla funzione. Questo meccanismo è noto con il nome di *passaggio di parametri per indirizzo*.
Consideriamo, per esempio, nel Listato 9.1 la funzione `scambia`, che ha l'effetto di scambiare il valore dei suoi parametri.

```
#include <stdio.h>

void scambia(int, int);

main()
{
  int x, y;

  x = 8;
  y = 16;
  printf("Prima dello scambio\n");
  printf("x = %d, y = %d\n", x, y);

  scambia(x, y);

  printf("Dopo lo scambio\n");
  printf("x = %d, y = %d\n", x, y);
}


/* Versione KO di scambia */
void scambia(int a, int b)
{
  int temp;

  temp = a;

  a = b;
  b = temp;
}
```

Listato 9.1 Il passaggio dei parametri per indirizzo

La chiamata di questa funzione non produce alcun effetto sui parametri attuali; cioè la chiamata

```
scambia(x, y);
```

non ha effetto sulle variabili intere `x` e `y`. Infatti i valori di `x` e `y` sono copiati nei parametri formali `a` e `b` e, quindi, sono stati scambiati i valori dei parametri formali, non i valori originali di `x` e `y`! Affinché `scambia` abbia un qualche effetto deve essere modificata in modo da ricevere gli indirizzi, anziché i valori, delle variabili (Listato 9.2). La relativa rappresentazione grafica del passaggio di parametri per indirizzo è data in Figura 9.2.

```
#include <stdio.h>

void scambia(int *, int *);
```

```
main()
{
  int x, y;

  x = 8;
  y = 16;
  printf("Prima dello scambio\n");
  printf("x = %d, y = %d\n", x, y);

  scambia(&x, &y);

  printf("Dopo lo scambio\n");
  printf("x = %d, y = %d\n", x, y);
}

/* Versione OK di scambia */
void scambia(int *a, int *b)
{
  int temp;

  temp = *a;
  *a = *b;
  *b = temp;
}
```

Listato 9.2 Ancora sullo scambio di valori

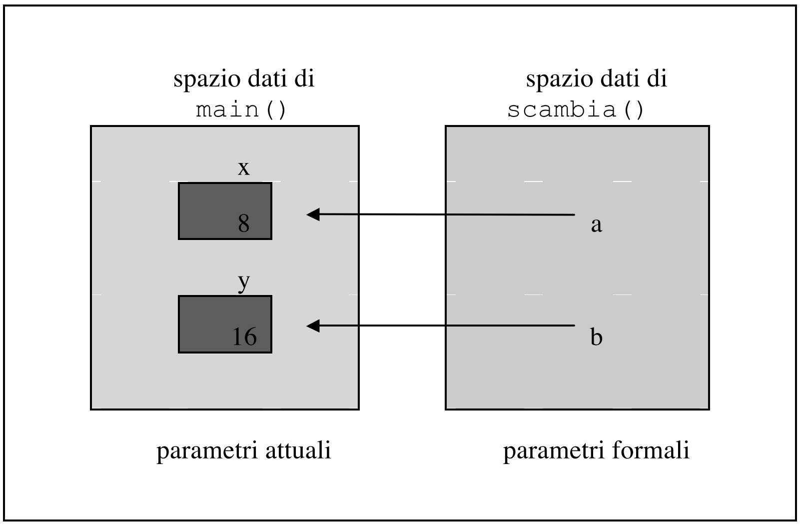


Figura 9.2 Passaggio di parametri per indirizzo

La simbologia `int *a` e `int *b`, usata nella definizione dei parametri formali, dichiara `a` e `b` come variabili di tipo puntatore a un intero. L'invocazione della funzione `scambia` deve essere modificata in modo da passare l'indirizzo delle variabili da scambiare:

```
scambia(&x, &y);
```

La medesima strategia del passaggio per valore di un indirizzo si può sfruttare con gli array (Listato 9.3).

```
#include <stdio.h>

char str[] = "BATUFFO";
int strlen(char *);
```

```
main()
{
printf("la stringa %s è lunga %d\n", str, strlen( str ));
}

int strlen( char *p)
{
  int i = 0;
  while (*p++) i++;
  return i;
}
```

Listato 9.3 Passaggio di un array

L'array dell'esempio è una stringa, cioè un array di `char` che termina con il carattere terminatore `\0`. Con l'inizializzazione

```
char str[] = "BATUFFO";
```

il primo elemento dell'array di caratteri `str` è inizializzato a puntare al primo elemento della costante di tipo stringa `"BATUFFO"`.
L'accorgimento di valutare una stringa per mezzo del puntatore `char *` è particolarmente utile nella scrittura di funzioni che manipolano stringhe. Nell'esempio la funzione `strlen` conta il numero di caratteri (escluso `\0`) di una stringa:

```
int strlen( char *p)
{
  int i = 0;
  while (*p++) i++;
  return i;
}
```

La funzione pone il contatore `i` a zero e comincia a contare caratteri finché non trova il carattere nullo. Una implementazione alternativa di `strlen` che usa la sottrazione di puntatori è:

```
int strlen( char *p )
{
  char *q = p;
  while ( *q++);
  return (q-p-1);
}
```

Le funzioni di manipolazione stringa gestiscono le stringhe sempre per mezzo di puntatori a carattere. Il C fornisce un vasto insieme di funzioni di manipolazione stringa, dichiarate nel file di *include* `<string.h>`. Perciò, per poter usare la libreria di manipolazione stringhe del C, occorre premettere la direttiva:

```
#include <string.h>
```

Si riportano di seguito le dichiarazioni delle più importanti funzioni di manipolazione stringa, alcune delle quali utilizzate nel Capitolo 5, allo scopo di riflettere sull'uso che viene fatto del passaggio dei parametri per indirizzo.

`char *strcat(char *string1, const char *string2);`
Concatena le stringhe *string1* e *string2* attaccando *string2* in coda a *string1*.

`char *strncat(char *string1, const char *string2, int n);`
Concatena le stringhe *string1* e *string2* attaccando *n* caratteri della stringa *string2* in coda a *string1*.

`int strcmp(const char *string1, const char *string2);`

Confronta *string1* con *string2*. Ritorna `0` se le stringhe sono identiche, un numero minore di zero se *string1* è minore di *string2*, e un numero maggiore di zero se *string1* è maggiore di *string2*.

`int strncmp(const char *string1, const char *string2, int n);`
Confronta i primi *n* caratteri di *string1* con *string2*. Ritorna `0` se le sottostringhe di *n* caratteri sono identiche, un numero minore di zero se *sottostring1* è minore di *sottostring2*, e un numero maggiore di zero se *sottostring1* è maggiore di *sottostring2*.

`char *strcpy(char *string1, const char *string2);`
Copia *string2* su *string1*.

`char *strncpy(char *string1, const char *string2, int n);`
Copia i primi *n* caratteri di *string2* su *string1*.

`int strlen(const char *string);`
Conta il numero di caratteri di *string*, escluso il carattere nullo.

`char *strchr(const char *string, int c);`
Ritorna il puntatore alla prima occorrenza in *string* del carattere `c`.

`char *strrchr(const char *string, int c);`
Ritorna il puntatore all'ultima occorrenza del carattere `c` nella stringa *string*.

`char *strpbrk(const char *string1, const char *string2);`
Ritorna un puntatore alla prima occorrenza della stringa *string2* in *string1*.

`int strspn(const char *string1, const char *string2);`
Trova la posizione del primo carattere in *string1* che non appartiene all'insieme di caratteri di *string2*.

`char *strtok(char *string1, const char *string2);`
Trova la prossima sequenza di caratteri (*token*) circoscritta dai caratteri *string2* nella stringa *string1*.

Il lettore può facilmente verificare l'uso di puntatori a `char` che si ha nelle funzioni di manipolazione stringa. Si osservi inoltre l'uso della parola chiave C `const`. Essa sta a indicare che, anche se l'indirizzo `char *` è passato alla funzione, la funzione non può andare a modificare le locazioni di memoria puntate da tale indirizzo, può solamente andare a leggere le locazioni puntate da quell'indirizzo. Con tale precisazione, semplicemente leggendo il prototype della funzione si capisce quali sono le stringhe che vengono modificate dalla corrispondente funzione di manipolazione.

9.5 Oggetti dinamici

I puntatori sono usati nella creazione e manipolazione di oggetti dinamici. Mentre gli oggetti statici vengono creati specificandoli in una definizione, gli oggetti dinamici sono creati durante l'esecuzione del programma. Il numero degli oggetti dinamici non è definito dal testo del programma, come per gli oggetti creati attraverso una definizione: essi vengono creati o distrutti durante l'esecuzione del programma, non durante la compilazione. Gli oggetti dinamici, inoltre, non hanno un nome esplicito, ma occorre fare riferimento a essi per mezzo di puntatori.
Il valore `NULL`, che può essere assegnato a qualsiasi tipo di puntatore, indica che nessun oggetto è puntato da quel puntatore. È un errore usare questo valore in riferimento a un oggetto dinamico.
Il puntatore `NULL` è un indirizzo di memoria che corrisponde al valore convenzionale di puntatore che non punta a nulla e la sua definizione può essere diversa da macchina a macchina. Per esempio:

```
#include <stdio.h>

main()
{
  char *p;
  ...
  p = NULL
  ...
  if (p != NULL {
  ...
  }
  else {
  ...
  }
}
```

Il valore di puntatore nullo `NULL` è una costante universale che si applica a qualsiasi tipo di puntatore (puntatore a `char`, a `int` ecc.). Generalmente la sua definizione è:

```
#define NULL 0
```

ed è contenuta in `<stdio.h>`.

Come detto, in C la memoria è allocata dinamicamente per mezzo delle funzioni di allocazione `malloc` e `calloc` che hanno le seguenti specifiche:

```
void *malloc(int num);  /* num: quantità di memoria da allocare */

void *calloc(int numele, int eledim);
                       /* numele: numero di elementi;  eledim:
                          quantità di memoria per ogni elemento */
```

Sia `malloc` sia `calloc` ritornano un puntatore a carattere che punta alla memoria allocata. Se l'allocazione di memoria non ha successo – o perché non c'è memoria sufficiente, o perché si sono passati dei parametri sbagliati – le funzioni ritornano il puntatore `NULL`.
Per stabilire la quantità di memoria da allocare è molto spesso utile usare l'operatore `sizeof`, che si applica nel modo seguente:

`sizeof(` *espressione* `)`

nel qual caso restituisce la quantità di memoria richiesta per memorizzare *espressione*, oppure

`sizeof(` *T* `)`

nel qual caso restituisce la quantità di memoria richiesta per valori di tipo *T*. Vediamo un esempio:

```
main()
{
  char a[10];
  int i;

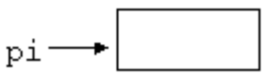
  i = sizeof(a[10]);
  printf("L'array %s ha dimensione = %d\n", a, i);

  i = sizeof(int);
  printf("Gli interi hanno dimensione %d", i);
}
```

L'operatore `sizeof` ritorna un intero maggiore di zero corrispondente al numero di `char` che formano l'espressione o il tipo. Si ricordi che l'unità di misura di `sizeof` è il `char` e non il byte, come potrebbe venire naturale pensare.
Gli allocatori `malloc` e `calloc` ritornano un puntatore all'oggetto dinamico creato. In realtà, gli allocatori ritornano dei puntatori a `void`, cioè a tipo generico, e perciò devono essere esplicitamente convertiti in un tipo specifico. Il valore che ritorna dagli allocatori di memoria è molto importante perché è solo attraverso di esso che si può far riferimento agli oggetti dinamici.
Consideriamo per esempio l'istruzione:

```
pi = (int *) malloc (sizeof(int));
```

che alloca una quantità di memoria sufficiente per accogliere un intero. Questo intero, allocato dinamicamente e di cui non si conosce il nome, può essere raggiunto per mezzo del puntatore `pi`. L'indirizzo dell'intero è assegnato a `pi` dopo aver esplicitamente convertito il tipo `void *`, ritornato `malloc`, nel tipo `int *`, il tipo della variabile `pi`, mediante la semplice espressione `(int *)` detta *cast*. Graficamente l'oggetto dinamico puntato da `pi` potrebbe essere rappresentato come segue.



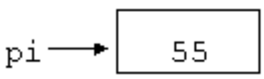
La scatola vuota simboleggia lo spazio riservato dall'intero. Per poter utilizzare le funzioni di allocazione è necessario includere la libreria `malloc.h` e/o `stdlib.h`; in implementazioni del C meno recenti la libreria da includere è stddef.h:

```
#include <malloc.h>
#include <stdlib.h>
```

Si accede a un oggetto dinamico tramite un puntatore e l'operatore di indirezione. Così, nell'esempio, si accede all'intero puntato da `pi` con il nome `*pi`. Per esempio:

```
*pi = 55;
```

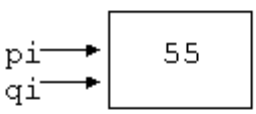
Graficamente l'effetto della precedente assegnazione può essere rappresentato come segue.



Lo stesso valore del puntatore può essere assegnato a più di una variabile puntatore. In tal modo si può far riferimento a un oggetto dinamico con più di un puntatore. Un oggetto cui si fa riferimento con due o più puntatori possiede degli *alias*. Per esempio, il risultato dell'assegnazione

```
qi = pi;
```

è di creare due puntatori allo stesso oggetto, cioè due alias. Graficamente l'effetto della precedente assegnazione può essere rappresentato come segue.



> ✓ **NOTA**
> Un uso smodato degli alias può deteriorare la leggibilità di un programma. Il fatto di accedere lo stesso oggetto con puntatori diversi può rendere difficoltosa l'analisi locale del programma, cioè la lettura di una porzione di codice senza avere in testa il tutto.

Gli oggetti dinamici devono essere esplicitamente deallocati dalla memoria se si vuole recuperare dello spazio. La deallocazione esplicita dello spazio di memoria avviene con la funzione `free`, così specificata:

```
free( char * ptr )
```

Se non si effettua questa operazione lo spazio di memoria verrà perso, cioè non sarà possibile riutilizzarlo. Per esempio:

```
free( pi );
```

libera la memoria occupata dall'intero puntato da `pi`. Occorre prestare molta attenzione a evitare errori del tipo: "fare riferimento a un oggetto che è già stato deallocato". Alcuni linguaggi, come il Lisp, lo Snobol, il Perl e Java, hanno dei meccanismi automatici di recupero della memoria detti "spazzini" (*garbage collector*) . In C il programmatore deve raccogliere la "spazzatura" da solo.

9.6 Indirizzamento assoluto della memoria

Il C è un linguaggio tipicamente usato per la programmazione di sistema, cioè per la programmazione di dispositivi hardware. Un classico problema della programmazione di sistema è l'indirizzamento diretto della memoria. Usando i puntatori e il cast è molto semplice fare riferimento a un indirizzo assoluto di memoria. Per esempio, supponiamo che `pt` sia un puntatore di tipo `T *`; questo puntatore lo si fa puntare alla locazione in memoria `0777000` nel seguente modo:

```
pt = (T *) 0777000;
```

Questa tecnica è comunemente usata nella costruzione di driver. Per esempio, nel caso del sistema operativo MS-DOS è pratica comune accedere direttamente alla memoria video per la gestione degli output su video. Occorre però tenere presente che la maggior parte dei sistemi operativi impedisce al programmatore la gestione diretta dell'hardware. È infatti il sistema operativo che offre l'interfaccia verso l'hardware, mettendo a disposizione della funzioni le cui invocazioni sono dette *chiamate di sistema*.

9.7 Gestione di una sequenza

Nell'ultimo paragrafo del Capitolo 7 abbiamo esaminato un programma per la gestione di una sequenza tramite un menu con le opzioni di immissione, ordinamento, ricerca completa e ricerca binaria. In quella sede l'array che conteneva la sequenza era una variabile generale cui tutte le funzioni accedevano direttamente.
Adesso presentiamo le modifiche necessarie perché il tutto avvenga mediante un array locale alla funzione `gestione_sequenza` e il passaggio del suo indirizzo alle altre funzioni. Innanzitutto le dichiarazioni devono essere fatte in modo da includere il parametro puntatore all'array:

```
int immissione( int, int * );
void ordinamento( int, int * );
int ricerca( int, int , int * );
int ric_bin( int, int , int * );
void visualizzazione( int, int * );
```

Alle dichiarazioni è stato aggiunto `int *`, per indicare che quel parametro sarà un puntatore a un oggetto di tipo `int`. Supponiamo che in `gestione_sequenza` venga definito l'array `sequenza`:

```
int sequenza[MAX_ELE];
```

Al momento della chiamata delle funzioni tale array deve essere passato come parametro attuale:

```
case 1: n = immissione( n, sequenza );
case 2: ordinamento( n, sequenza );
posizione = ricerca( n, ele, sequenza );
posizione = ric_bin( n, ele, sequenza );
case 5: visualizzazione( n, sequenza );
```

Nella definizione delle funzioni, a sua volta, deve essere esplicitato un nuovo parametro formale:

```
int immissione( int n, int *vet )
void ordinamento( int n, int *vet )
int ric_bin( int n, int ele, int *vet )
void visualizzazione( int n, int *vet )
```

Sorprendentemente, all'interno di ogni funzione, non cambia niente: infatti `vet` è una variabile puntatore all'array: i sottoprogrammi possono accedere all'array e modificare il suo contenuto ma, essendo `vet` una variabile locale, non possono modificarlo.

9.8 Esercizi

[*Nota: risolvere i seguenti problemi utilizzando i puntatori*.]

*1. Scrivere un programma che esegua la scansione e la visualizzazione di un vettore di interi.

*2. Scrivere un programma che esegua la scansione e la visualizzazione di un vettore di stringhe.

*3. Scrivere una funzione che ritorni un puntatore alla prima occorrenza della stringa `t` in una stringa `s`. Se la stringa `t` non è contenuta in `s` allora la funzione ritorna un puntatore `NULL`.

*4. Scrivere almeno tre differenti versioni di una funzione che effettui la copia di una stringa su un'altra.

*5. Scrivere un programma che prenda in ingresso la dimensione di un buffer e la allochi dinamicamente.

*6. Modificare il programma `gestione_sequenza` cui si fa riferimento nell'ultimo paragrafo di questo capitolo in modo che la funzione di immissione sequenza non ritorni nessun valore ma abbia in ingresso il puntatore alla variabile intera `n` di `gestione_sequenza` per poterla modificare.

10.1 Iterazione e ricorsione

Quando si vuole ripetere l'esecuzione di un gruppo di istruzioni, un'alternativa alle strutture iterative come `for` o `while` è rappresentata dalle *funzioni ricorsive*. Una funzione si dice ricorsiva se chiama se stessa direttamente o indirettamente.

Per alcune classi di problemi le soluzioni ricorsive sono eleganti, sintetiche e più chiare delle altre. Un esempio di questo fatto si può trovare nel calcolo del fattoriale, esaminato nel Capitolo 3. Ricordiamo che il fattoriale $n!$ del numero n, dove n è un intero maggiore o uguale a 2, è dato da:

$$n! = n\cdot(n-1)\cdot(n-2)\cdot(n-3)\cdot\ldots\cdot2\cdot1$$

Inoltre 0! e 1! sono per definizione uguali a 1.

Il fattoriale è ricorsivo per definizione. Può essere espresso come

$$n! = n\cdot(n-1)!$$

ovvero il fattoriale di n è uguale a n moltiplicato per il fattoriale di $n-1$; ricorsivamente, 4! è dunque uguale a 4 moltiplicato per il fattoriale di (4−1), e così via.

Se `fat` è la funzione che calcola il fattoriale, dovrà allora ricevere in ingresso il numero intero su cui operare e dovrà restituirne il fattoriale (Figura 10.1).

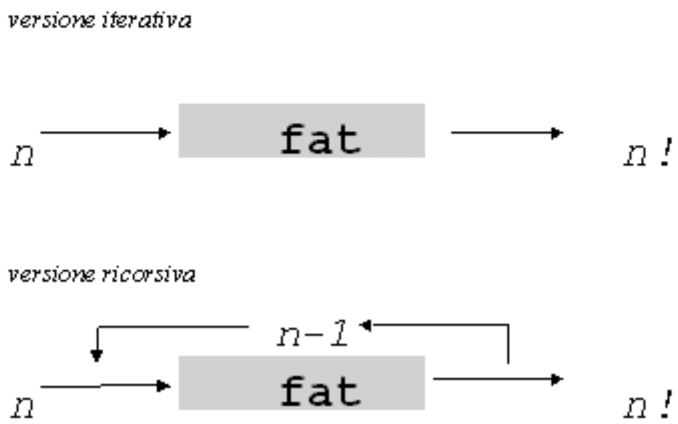


Figura 10.1 La funzione `fat` restituisce il fattoriale di `n`

Per restituire il fattoriale al programma chiamante la funzione utilizzerà l'istruzione `return`:

```
return(n*fat(n-1));
```

che ritorna il valore di `n` moltiplicato per il fattoriale di `n-1`. Il calcolo del fattoriale di `n-1` lo si ottiene invocando ancora una volta la stessa funzione `fat` e passandole come argomento `n-1`; in questo modo si ottiene l'iterazione. Il ciclo, a un certo punto, deve terminare, per cui è necessaria una condizione di fine, che impostiamo così:

```
if(n==0)
       return(1);
     else
       return(n*fat(n-1));
```

Quando il valore passato alla funzione è uguale a 0, non ci sono valori da considerare e `fat` ritorna 1 (0!).
Confrontiamo dunque la funzione ricorsiva `fat` con il programma iterativo del Capitolo 3.

procedura ricorsiva   procedura iterativa

```
-----------------------------------------------------------
fat(int n)
{
if(n==0)                         if(n==0)
  return(1);                       fat = 1;
else                             else
  return(n*fat(n-1));              for(fat=n; n>2; n--)
}                                    fat = fat*(n-1);
```

La soluzione ricorsiva corrisponde direttamente alla definizione di fattoriale. Nel Listato 10.1 viene presentato un programma completo per il calcolo del fattoriale.

```
/* Calcolo del fattoriale con una funzione ricorsiva */

#include <stdio.h>

fat(int);

main()
{
int n;

printf("CALCOLO DI n!\n\n");
printf("Inser. n: \t");
```

```
scanf("%d", &n);
printf("Il fattoriale di: %d ha valore: %d\n", n, fat(n));
}

fat(int n)
{
if(n==0)
  return(1);
else
  return(n*fat(n-1));
}
```

Listato 10.1 Calcolo del fattoriale mediante una funzione ricorsiva

Esaminiamo ora più da vicino gli ambienti che la funzione ricorsiva genera a ogni sua chiamata, prendendo l'esempio del calcolo di 4! (Figura 10.2).

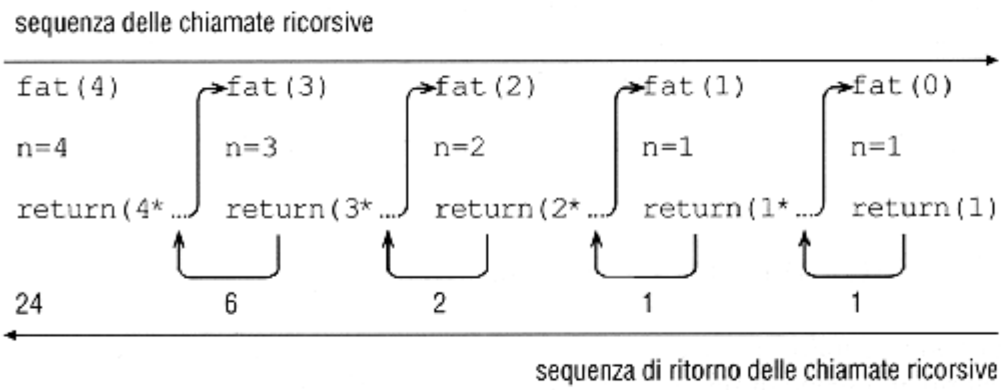


Figura 10.2 Ambienti creati dalla funzione `fat` con ingresso `n`=4

Osservando la figura possiamo vedere che a ogni chiamata di **`fat`** viene creata una nuova variabile `n` locale a quell'ambiente. Quando termina il ciclo delle chiamate, ogni ambiente aperto si chiude e passa all'ambiente precedente il valore calcolato.

✓ **NOTA**
La zona di memoria riservata alle chiamate viene gestita con la logica di una pila, concetto che tratteremo in modo specifico più avanti, quando parleremo di strutture dati. A ogni invocazione di `fat`*, il sistema alloca uno spazio di memoria libero in testa alla pila riservato al suo parametro formale* `n`*. In Figura 10.2 si osserva come la sequenza di chiamate e il ritorno delle stesse vengano gestiti come una pila, in cui l'ultimo elemento creato è il primo a essere eliminato. Lo spazio di memoria allocato fa parte dell'ambiente locale a ogni chiamata di* `fat`*.*

Avendo dichiarato `n` e `fat` di tipo `int` (si ricordi che quando nella dichiarazione di funzione non ne viene specificato il tipo, viene implicitamente assunto `int`) si ottengono risultati significativi con valori di `n` piuttosto bassi; per aumentare questo limite si può utilizzare il tipo `long int`, che ha dimensione maggiore o uguale a un `int`. Dobbiamo specificarlo in fase dichiarativa, all'inizio del file:

```
long int fat(long int);
```

nella definizione della funzione:

```
long int fat(long int n) {...};
```

e dobbiamo modificare la `printf` nel programma principale, in modo da indicare il formato in cui si desidera la visualizzazione (`%ld`) di `fat`:

```
printf("Il fattoriale di: %d ha valore: %ld\n", n, fat(n));
```

Se poi si desidera calcolare fattoriali ancora più alti si deve usare una funzione di tipo `float` o `double`. Dato che il risultato è sempre un intero, è meglio specificare nella `printf` di non visualizzare cifre dopo la virgola:

```
printf("Il fattoriale di: %d ha valore: %.0f\n", n, fat(n));
```

10.2 Permutazione e disposizioni

In questo paragrafo e nel successivo considereremo alcune procedure di calcolo combinatorio allo scopo di scrivere interessanti funzioni ricorsive ▬.

Si definiscono *permutazioni semplici* di *n* oggetti distinti i gruppi che si possono formare in modo che ciascuno contenga tutti gli *n* oggetti dati e che differisca dagli altri soltanto per l'ordine in cui vi compaiono gli oggetti stessi. Dati due oggetti `e1`, `e2` si possono avere solamente due permutazioni; se gli oggetti sono tre le permutazioni semplici diventano sei; se gli oggetti sono quattro (`e1 e2 e3 e4`), si hanno le seguenti 24 possibilità:

```
e1 e2 e3 e4    e1 e2 e4 e3     e2 e1 e3 e4     e1 e2 e4 e3
     e3 e1 e2 e4    e3 e1 e4 e2     e1 e3 e2 e4     e1 e3 e4 e2
     e2 e3 e1 e4    e2 e3 e4 e1     e3 e2 e1 e4     e3 e2 e4 e1
     e1 e4 e2 e3    e1 e4 e3 e2     e2 e4 e1 e3     e2 e4 e3 e1
     e3 e4 e1 e2    e3 e4 e2 e1     e4 e1 e2 e3     e4 e1 e3 e2
     e4 e2 e1 e3    e4 e2 e3 e1     e4 e3 e1 e2     e4 e3 e2 e1
```

In generale, il numero di permutazioni *P* di *n* oggetti è dato $P_n = n!$, da cui risultano appunto, nel nostro caso, 4! = 24 possibilità distinte.

Questo è un problema che abbiamo già risolto. Se desideriamo conoscere il numero di permutazioni di 13 oggetti è sufficiente mandare in esecuzione l'ultimo programma del paragrafo precedente, con l'accortezza di utilizzare un tipo dati adeguato, per scoprire che sono: 6 227 020 800.

Dati *n* oggetti distinti e detto *k* un numero intero positivo minore o uguale a *n*, si chiamano invece *disposizioni semplici* di questi *n* oggetti i gruppi distinti che si possono formare in modo che ogni gruppo contenga soltanto *k* oggetti e che differisca dagli altri o per qualche oggetto, o per l'ordine in cui gli oggetti stessi sono disposti. Le disposizioni di quattro oggetti (*n*=4) presi uno a uno (*k*=1) sono dunque i gruppi che contengono un solo oggetto:

```
e1      e2      e3      e4
```

cioè in totale quattro. Le disposizioni di quattro oggetti presi due a due (*k*=2) sono invece 12:

```
e1 e2      e2 e1      e3 e1      e4 e1
     e1 e3      e2 e3      e3 e2      e4 e2
     e1 e4      e2 e4      e3 e4      e4 e3
```

Il calcolo delle disposizioni usa la formula generale:

$D_{n,k} = n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n-k+2) \cdot (n-k+1)$

Nel caso di *n*=4 e *k*=1 verifichiamo

$D_{4,1} = 4$

e nel caso di *n*=4 e *k*=2

$D_{4,2} = 4 \cdot 3 = 12$

Possiamo dunque scrivere una procedura ricorsiva che calcoli le disposizioni semplici:

```
int dispo(int k, int n, int m)
     {
     if(n==m-k)
       return(1);
     else
       return(n*dispo(k, n-1, m));
```

```
}
```

Al momento della prima invocazione di `dispo`:

```
dispo(k, n, n);
```

dobbiamo passare alla funzione, oltre ai valori di `k` e di `n`, anche un ulteriore valore `n` (che diventa il parametro formale `m`) perché essa possa conoscere il numero totale degli oggetti e quindi effettuare il controllo `n=m-k`, dato che a ogni ulteriore chiamata il parametro formale `n` viene decrementato di una unità rispetto al precedente. Naturalmente si poteva anche optare per l'utilizzo di una variabile globale `m` inizializzata al valore di `n` (si veda il Listato 10.2).

```
/* Calcolo delle disposizioni semplici di n oggetti presi k a k */

#include<stdio.h>

int dispo(int, int, int);

main()
{
int n, k;

printf("Disposizioni semplici di k su n oggetti\n");
printf("Inser. n: \t");
scanf("%d", &n);
printf("Inser. k: \t");
scanf("%d", &k);
printf("Le dispos. sempl. di %d su %d sono: %d\n", k, n, dispo(k, n, n));
}

int dispo(int k, int n, int m)
{
if(n==m-k)
  return(1);
else
  return(n*dispo(k, n-1, m));
}
```

Listato 10.2 Calcolo delle disposizioni semplici

Osserviamo che il calcolo delle disposizioni è simile a quello del fattoriale. In particolare, le disposizioni di *n* elementi presi *n* a *n* sono proprio pari a *n*!:

$D_{n,n} = n!$

Nel caso fosse *n*=4 e *k*=4 avremmo:

$D_{4,4} = 24$

Possiamo allora scrivere una nuova funzione `dispo2()` che sfrutti la funzione `fat` precedentemente definita:

```
/* Calcolo delle disposizioni semplici utilizzando la funzione
       per il calcolo delle permutazioni */

    int dispo2(int k, int n)
    {
    return(fat(n)/fat(n-k));
    }
```

Infatti $D_{n,k}$ = `fat(n)/fat(n-k)`, come si può facilmente dedurre dal confronto delle due formule.

10.3 Combinazioni

Si chiamano *combinazioni semplici* di n oggetti distinti, presi k a k ($k \leq n$) i gruppi di k oggetti che si possono formare con gli n oggetti dati, in modo che i gruppi stessi differiscano tra loro almeno per un oggetto. Per esempio, i quattro elementi `e1`, `e2`, `e3` ed `e4`, presi due a due, danno origine alle seguenti sei combinazioni:

```
e1 e2    e1 e3    e1 e4     e2 e3     e2 e4    e3 e4
```

La formula generale che consente di calcolare il numero delle combinazioni è

$C_{n,k} = D_{n,k}/k!$

Dunque il numero di combinazioni di n oggetti presi k a k è uguale al numero di disposizioni di n oggetti presi k a k, diviso k fattoriale.

La funzione `comb`, che calcola il numero di combinazioni semplici possibili, può richiamare `dispo` per calcolare le disposizioni $D_{n,k}$ e `fat` per calcolare il fattoriale, passando `k` come numero di elementi:

```
comb(int k, int n)
     {
     return(disp(k, n)/fat(k));
     }
```

Nel Listato 10.3 viene presentato il programma relativo al calcolo delle combinazioni semplici.

```
/* Calcolo delle combinazioni semplici di n oggetti presi k a k */

#include <stdio.h>

int comb(int, int);
int dispo(int , int, int);
int fat(int);

main()
{
int n, k;

printf("Combinazioni semplici di k su n oggetti\n");
printf("Inserire n: \t");
scanf("%d", &n);
printf("Inserire k: \t");
scanf("%d", &k);
printf("Le combin. sempl. di %d su %d sono: %d\n",k,n,comb(k,n));
}

comb(int k, int n)
{
return(dispo(k, n, n)/fat(k));
}

int dispo(int k, int n, int m)
{
if(n==m-k)
  return(1);
else
  return(n*dispo(k, n-1, m));
}

fat(int n)
{
```

```
if(n==0)
  return(1);
else
  return(n*fat(n-1));
}
```

Listato 10.3 Calcolo delle combinazioni semplici; vengono utilizzate le funzioni `dispo` e `fat` viste precedentemente

Una prima alternativa è quella di utilizzare in `comb` soltanto la funzione `dispo`, dato che $D_{k,k}$ è uguale a k!:

```
/* Calcolo delle combinazioni semplici
        utilizzando soltanto la funzione per il
        calcolo delle disposizioni              */

     comb(int k, int n)
     {
     return(dispo(k, n, n)/dispo(k, k, k));
     }
```

Una seconda possibilità si ottiene sfruttando la funzione `dispo2` che, come abbiamo visto in precedenza, utilizzava a sua volta `fat` per calcolare le disposizioni:

```
/* Calcolo delle combinazioni semplici utilizzando
        dispo2() e fat()                                */

     comb(int k, int n)
     {
     return(dispo2(k, n, n)/fat(k));
     }

     /* Calcolo delle disposizioni semplici utilizzando fat() */

     int dispo2(int k, int n)
     {
     return(fat(n)/fat(n-k));
     }
```

Così facendo abbiamo decomposto le formule risolutive di combinazioni e disposizioni rimandando il problema al calcolo del fattoriale. Attenzione, comunque: se n e k superano un certo valore, che dipende dalla dimensione degli `int` e dei `long int` dello specifico compilatore, si devono utilizzare funzioni e parametri di tipo `float`.

10.4 La successione di Fibonacci

Nella particolare successione detta di Fibonacci ogni termine è ottenuto sommando i due che lo precedono; il termine generico è pertanto

$F(n) = F(n-1) + F(n-2)$

dove n è un numero intero maggiore o uguale a 2. Inoltre $F(0) = 0$ e $F(1) = 1$. Dunque si ha:

```
F(2) = F(2-1) + F(2-2) = F(1) + F(0) = 1+0 = 1
     F(3) = F(3-1) + F(3-2) = F(2) + F(1) = 1+1 = 2
     F(4) = F(4-1) + F(4-2) = F(3) + F(2) = 2+1 = 3
     F(5) = F(5-1) + F(5-2) = F(4) + F(3) = 3+2 = 5
     F(6) = F(6-1) + F(6-2) = F(5) + F(4) = 5+3 = 8
     F(7) = F(7-1) + F(7-2) = F(6) + F(5) = 8+5 = 13
     ...
```

È evidente il carattere ricorsivo di tale definizione (si veda il Listato 10.4): ogni chiamata di `fibo` genera due ulteriori chiamate ricorsive, una passando il parametro attuale `n-1`, l'altra `n-2`; dunque al crescere del valore di *n* la memoria tende rapidamente a saturarsi.

```
/* Calcolo dei numeri di Fibonacci */

#include <stdio.h>

long int fibo(int);

main()
{
int n;

printf("Successione di Fibonacci f(0)=1 f(1)=1 f(n)=f(n-1)+f(n-2)");
printf("\nInserire n: \t");
scanf("%d", &n);
printf("Il termine della successione di argomento %d è: %d\n", n, fibo(n));
}

long int fibo(int n)
{
if(n==0)       return(0);
else if(n==1)  return(1);
     else      return(fibo(n-1)+fibo(n-2));
}
```

Listato 10.4 Calcolo della successione di Fibonacci

✓ **NOTA**
I programmi che sfruttano la ricorsività sono in larga misura inefficienti, tanto in termini di occupazione di memoria quanto in termini di velocità di esecuzione. La ragione principale è che nelle funzioni ricorsive spesso vengono ripetuti calcoli già eseguiti in precedenza. Si riprenda l'esempio delle combinazioni: nel caso di `comb(5,2)`, `comb(2,1)` è calcolato tre volte!
Si può facilmente verificare il peso di queste ripetizioni inserendo, all'interno delle funzioni esaminate, prima della chiamata ricorsiva, una `printf` che visualizzi il valore delle variabili trattate dalle funzioni stesse.

10.5 Ordinamento con *quicksort*

Nel Capitolo 5 abbiamo esaminato due metodi di ordinamento: quello ingenuo e *bubblesort*; quest'ultimo è stato successivamente utilizzato in più occasioni. Come ulteriore e più complesso esempio di procedura ricorsiva consideriamo ora il principe degli algoritmi di ordinamento, *quicksort* (letteralmente: "ordinamento veloce"). Con metodi matematici è possibile dimostrare che quicksort è in media il più veloce metodo di ordinamento a uso generale.

Uno degli aspetti più interessanti di quicksort è che esso ordina gli elementi di un vettore seguendo una procedura analoga a quella seguita da una persona per ordinare un insieme di oggetti. Immaginiamo per esempio di dover mettere in ordine su uno scaffale 300 numeri di una rivista di informatica. In genere, magari in maniera non sempre cosciente, si procede in questo modo: preso un primo fascicolo si portano alla sua sinistra tutti i numeri più piccoli di questo e alla sua destra quelli più grandi. Sui due gruppi così ottenuti si procede poi allo stesso modo ottenendo via via insiemi sempre più piccoli e facilmente ordinabili. Alla fine del processo la raccolta risulta ordinata.

Quicksort agisce essenzialmente allo stesso modo: prima crea due grossi blocchi che poi inizia a ordinare costruendone altri sempre più piccoli, per ritrovarsi alla fine una sequenza interamente ordinata.

L'algoritmo di quicksort inizia determinando un ipotetico valore medio del vettore, detto *pivot* ("pernio"), quindi suddivide gli elementi in due parti: quella degli elementi più piccoli e quella degli elementi più grandi del pivot. Non è indispensabile che la suddivisione sia esattamente in parti uguali: l'algoritmo funziona con qualunque approssimazione. Tuttavia, quanto più la suddivisione è esatta, tanto più l'ordinamento risulta veloce.

```c
/* Ordinamento quicksort  di un array di int */

#include <stdio.h>

#define N 10 /* numero elementi dell'array */
int v[N];    /* array contenente gli interi immessi */

void quick(int, int);
void scambia(int *, int *);

main()
{
int i;
for(i=0; i<N; i++) {
  printf("\nImmettere un intero n.%d: ",i);
  scanf("%d", &v[i]);
}

quick(0,N-1);       /* Chiamata della procedura quick */

for(i=0; i<N; i++)      /* Sequenza ordinata */
  printf("\n%d", v[i]);
putchar('\n');
}

/* Procedura ricorsiva "quick" */
void quick(int sin, int des)
{
  int i, j, media;
  media= (v[sin]+v[des]) / 2;
  i = sin;
  j = des;

  do {
    while(v[i]<media) i = i+1;
    while(media<v[j]) j = j-1;
    if(i<=j) {
      scambia(&v[i], &v[j]);
      i = i+1;
      j = j-1;
    }
  }
  while (j>=i);

if(sin<j) quick(sin, j);  /* Invocazione ricorsiva */
if(i<des) quick(i, des);  /* Invocazione ricorsiva */
}

void scambia(int *a, int *b)
{
  int temp;
```

```
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Listato 10.5 Ordinamento di una sequenza con il metodo quicksort

Il programma del Listato 10.5 realizza l'ordinamento quicksort su un vettore di 10 elementi interi; nella Figura 10.3 viene illustrata un'applicazione della procedura `quick`, dove la prima colonna contiene la sequenza iniziale. La stima del pivot avviene, in modo non molto raffinato, facendo semplicemente la media tra il primo e l'ultimo elemento della parte del vettore su cui l'algoritmo sta lavorando. Una volta stimato il pivot, la procedura sposta tutti gli elementi di valore minore di questo nella parte bassa del vettore, tutti quelli con valore maggiore nella parte alta. A ogni passo del processo quicksort ordina quindi attorno al pivot gli elementi del blocco del vettore esaminato. Man mano che i blocchi diventano sempre più piccoli il vettore tende a essere completamente ordinato.

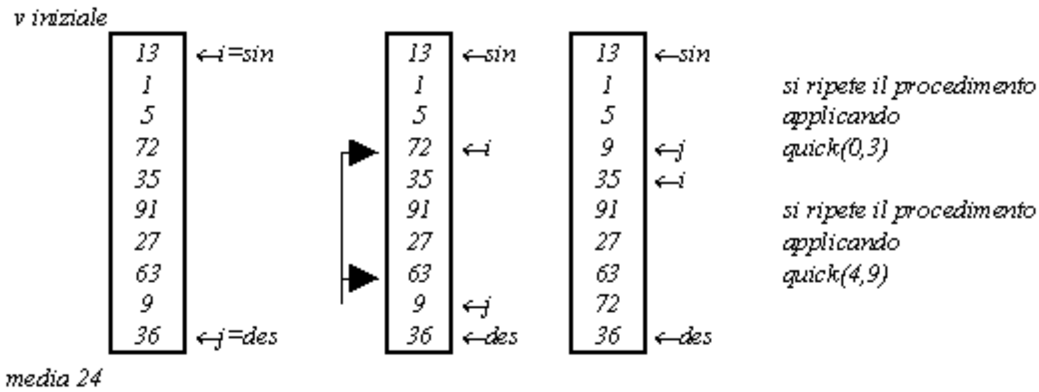


Figura 10.3 Fasi di lavoro di quicksort applicato a un vettore di 10 interi

La procedura `quick` del Listato 10.5 viene inizialmente applicata sull'intero vettore (dall'elemento di indice 0 a quello di indice *N*–1); è calcolato il valore medio `media` tra `vet[sin]` e `vet[des]` che alla prima chiamata è 24, la media appunto tra i due estremi del vettore: 13 (`vet[0]`) e 36 (`vet[9]`) (Figura 10.3). Successivamente `quick` scorre gli elementi del blocco a partire dal basso, fintantoché si mantengono minori della media:

```
while(v[i]<media) i = i+1;
```

Analogamente, la procedura scorre gli elementi del blocco a partire dall'alto, fintantoché si mantengono maggiori della media:

```
while(media<v[j]) j = j-1;
```

A questo punto, se l'indice dell'elemento della parte inferiore del blocco è minore o uguale all'indice dell'elemento della parte superiore, si effettua lo scambio tra i rispettivi valori:

```
if(i<=j) {scambia(&v[i], &v[j]); ...
```

Nel caso in esame viene scambiato il quarto elemento (`i=3`) con il nono (`j=8`), come mostrato nella seconda colonna della Figura 10.3. Successivamente la procedura incrementa gli indici degli elementi considerati e prosegue le scansioni dal basso e dall'alto finché `i` non risulti maggiore di `j`, condizione controllata dal `do-while`. Nell'esempio, dopo la seconda iterazione `i` assume valore 4 e `j` valore 5, così che il ciclo `do-while` ha termine. Quindi la procedura richiama se stessa passando nuovi valori superiori e inferiori e analizzando blocchi sempre più piccoli del vettore:

```
if(sin<j) quick(sin, j);
if(i<des) quick(i, des);
```

Nel caso in esame `sin` vale 0 e `j` vale 3, per cui è chiamata `quick(0,3)` e il blocco che viene ordinato è quello tra `vet[0]`, che vale 13, e `vet[3]`, che vale 9.

Quando raggiunge il livello più basso, ovvero quando i sottoinsiemi trattati sono costituiti da un solo elemento, la ricorsione termina e la procedura restituisce al programma il vettore ordinato.

È importante notare come l'uso di procedure ricorsive non sia, nella scrittura di quicksort come in quella di ogni altro algoritmo, indispensabile. Ciò nonostante, una versione non ricorsiva di quicksort risulta laboriosa e di difficile lettura, così che anche in questo caso, come nel calcolo del fattoriale, si preferisce in genere scrivere il programma nella sua versione ricorsiva.

Nei capitoli precedenti avevamo esaminato il metodo di ordinamento *bubblesort*, adesso abbiamo visto quicksort; il parametro rispetto al quale sono confrontati gli algoritmi di ordinamento è dato dal numero di confronti necessari per ordinare un vettore, poiché questo risulta essere direttamente proporzionale al tempo impiegato dall'algoritmo. Tale numero è espresso come funzione del numero di elementi del vettore. Per esempio, nell'ordinamento di un vettore di *n* numeri casuali, se *n* è pari a 100, 500 o 1000, il numero stimato di confronti in media corrisponderà rispettivamente per bubblesort a 4950, 124.750 e 499.500; con quicksort si avranno 232, 1437 e 3254 confronti. Sottolineiamo che si tratta di *stime*, perché il numero effettivo dei confronti dipende comunque dai particolari valori del vettore da ordinare.

Questa valutazione è stata ottenuta in base a considerazioni matematiche di tipo teorico ma molti programmatori trovano più semplice confrontare i metodi di ordinamento misurando direttamente il tempo impiegato da ognuno di loro per ordinare il medesimo vettore. Anche in questo caso si mantiene la stessa classifica nelle prestazioni; per esempio, con 1000 elementi bubblesort è in media 70 volte più lento di quicksort. Notiamo ovviamente che il tempo assoluto occorrente dipende dall'elaboratore su cui si sta lavorando.

Il vantaggio della ricorsione risiede nella possibilità di realizzare algoritmi sintetici ed eleganti. Inoltre, la scrittura di funzioni ricorsive permette al programmatore di verificare a fondo le proprie conoscenze e possibilmente di migliorarle. Nei capitoli successivi vedremo l'uso della ricorsività nell'ambito delle strutture dati, come liste e alberi, dove a volte la soluzione migliore, per la natura stessa dei problemi che si affronteranno, sarà appunto quella ricorsiva.

10.6 Mutua ricorsività

Concludiamo questo capitolo introducendo brevemente il concetto di *mutua ricorsività*.

In una funzione `x` può essere presente una chiamata a una funzione `y`. Se `y` contiene a sua volta una chiamata a `x`, si ha una *ricorsività indiretta*. In generale, in questo processo di mutua ricorsività possono essere coinvolte più di due funzioni che indirettamente invocano se stesse tramite una successione di chiamate: dalla funzione `x` viene chiamata `y`, da `y` viene chiamata `t`, da `t` viene chiamata `z`, da `z` ancora `x` ecc. Nella Figura 10.4 viene mostrato un esempio di ricorsività indiretta.

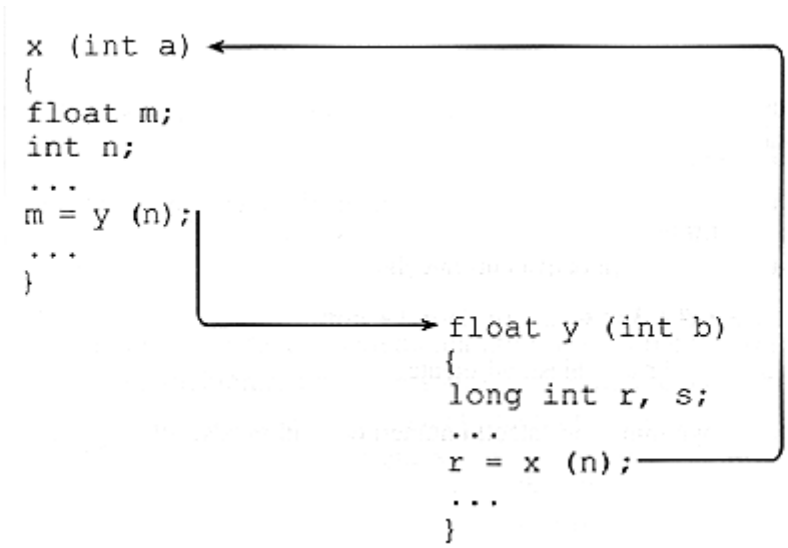


Figura 10.4 Esempio di ricorsività indiretta

10.7 Esercizi

1. Modificare la funzione ricorsiva che calcola il fattoriale in modo che venga stampato il valore dei fattoriali di tutti i numeri minori o uguali a n.

2. Scrivere una funzione ricorsiva che accetti in ingresso n valori e ne restituisca la somma al programma chiamante.

* 3. Predisporre una funzione che, in forma iterativa, calcoli la potenza: `base` elevata a `esponente`, dove `esponente` è un numero intero maggiore o uguale a zero.

4. Risolvere l'Esercizio 3 con una funzione ricorsiva.

* 5. Estendere la soluzione dell'Esercizio 3 in modo che `esponente` possa essere un numero intero qualsiasi (anche negativo).

* 6. Risolvere l'Esercizio 5 con una funzione ricorsiva.

* 7. Scrivere una funzione ricorsiva che calcoli il *massimo comune divisore* di due numeri interi positivi utilizzando l'algoritmo euclideo per cui:

$$\mathtt{MCD}(t,k) = \begin{cases} t & \text{se } k = 0 \\ \mathtt{MCD}(k,t) & \text{se } k > t \\ \mathtt{MCD}(k,\ t\%k) & \text{altrimenti} \end{cases}$$

8. Scrivere una funzione ricorsiva che calcoli il *massimo comune divisore* di due numeri interi positivi ricordando che

$$\mathtt{MCD}(t,k) = \begin{cases} \mathtt{MCD}(t-k,\ k) & \text{se } t>k \\ t & \text{se } t=k \\ \mathtt{MCD}(k,\ t) & \text{se } t<k \end{cases}$$

9. Confrontare gli algoritmi dei due precedenti esercizi e dire qual è più veloce; motivare la risposta.

10. Scrivere una versione ricorsiva della ricerca binaria su un vettore ordinato.

11. Scrivere una funzione ricorsiva che calcoli

$f(x,n) = 1 \cdot x + 2 \cdot x^2 + 3 \cdot x^3 + \dots + (n-1) \cdot x^{n-1} + n \cdot x^n$

con x `float` e n `int` richiesti all'utente.

12. Scrivere un programma che calcoli i numeri ottenuti in base alla seguente definizione:

$a_1 = 3$;
$a_2 = 7$;
$a_n = 2 \cdot a_{n-1} - 3 \cdot a_{n-2}$ per $n \geq 3$

Cosa si può dire a proposito del segno (positivo o negativo) dei valori di a_n?

11.1 Tipi e variabili

Un *tipo* rappresenta un insieme di valori e di operazioni che possono essere svolte su quei valori. Una *variabile* è una entità che conserva un valore del corrispondente tipo; inserire un nuovo valore in una variabile significa distruggere il vecchio valore presente in essa.
In C vi sono due categorie di tipi: *fondamentali* e *derivati*. I tipi fondamentali sono:

```
char    int    enum    float    double
```

Questi vengono detti anche tipi aritmetici, poiché corrispondono a valori che possono essere interpretati come numeri. I tipi derivati, invece, sono costruiti a partire dai tipi fondamentali, e sono:

```
void    array    funzioni    puntatori    strutture    unioni
```

Alcuni tra i tipi fondamentali e derivati sono già stati trattati. In questo capitolo riprenderemo le definizioni precedentemente introdotte e le completeremo aggiungendo i qualificatori di tipo, definendo il tipo `enum` e un nuovo insieme di possibili operazioni.
Prima di procedere alla descrizione dettagliata dei tipi fondamentali è necessario richiamare alcuni concetti di carattere generale, quali la dichiarazione e la definizione di un nome.
In C un qualunque nome deve essere dichiarato prima di poter essere usato all'interno del programma; dichiarare un nome significa specificare a quale tipo appartiene quel nome e quindi, indirettamente, quanta memoria dovrà essere riservata per quel nome, e quali sono le operazioni ammesse su di esso. Esempi di dichiarazione di un nome sono:

```
char   c;
int    conta = 0;
double pot(float, int);
char   *animale = "Anatra";
double cubo(float c) { return c*c*c }
extern int codice_errore;
```

Alcune di queste dichiarazioni sono anche delle definizioni, cioè hanno l'effetto di creare la relativa regione di memoria, detta "oggetto"; esattamente si tratta di:

```
char   c;
int    conta = 0;
char   *animale = "Anatra";
double cubo(float c) { return c*c*c }
```

Le prime due riservano una determinata zona di memoria della dimensione di un `char` e di un `int` cui associare rispettivamente il nome `c` e il nome `conta`. L'ultima associa al nome `cubo` una porzione di programma, corrispondente alle istruzioni che formano la funzione `cubo`. Le dichiarazioni:

```
double pot(float, int);
extern int codice_errore;
```

invece non corrispondono ad alcuna allocazione di memoria. Entrambe le istruzioni introducono un nome, ma rimandano la definizione del nome, cioè l'allocazione dell'oggetto corrispondente, a un'altra parte del programma. Nel Capitolo 7 abbiamo già visto la differenza tra dichiarazione e definizione di una funzione; questa differenza può esistere anche tra variabili che non siano funzioni, come nel caso della variabile `codice_errore` il cui tipo è `int`, e la cui definizione è `extern` (esterna) cioè definita in un qualche altro file, diverso da quello in cui è dichiarata per mezzo dell'istruzione .

```
extern int codice_errore;
```

Si ricordi che tutte le dichiarazioni che specificano un valore per il nome che introducono sono sempre anche delle definizioni. Per esempio:

```
int    conta = 0;
char   *animale = "Anatra";
double cubo(float c) { return c*c*c }
```

sono delle definizioni poiché accanto ai nomi `conta`, `animale` e `cubo` vengono specificati dei valori. Il valore iniziale associato a `conta` e `animale` può essere cambiato nel corso del programma, poiché si tratta di inizializzazioni; il valore associato alla funzione `cubo`, ossia il suo blocco di istruzioni, è permanente, cioè non può più essere cambiato nel corso del programma.

✓ **NOTA**
È molto importante che il programmatore C acquisisca piena padronanza dell'uso della memoria dell'elaboratore; deve quindi saper correttamente distinguere tra dichiarazioni che definiscono un'entità e dichiarazioni che introducono un nome senza definire l'entità corrispondente.

11.2 Tipi fondamentali

Il linguaggio C ha un insieme di tipi fondamentali associati alle principali unità di memoria di un calcolatore, e alle corrispondenti modalità d'uso più comuni. Così gli interi sono rappresentati dai tipi

```
char        int
```

mentre i numeri in virgola mobile sono rappresentati da

```
float   double
```

Il tipo `char` è un intero che generalmente ha una dimensione pari a un byte (8 bit). Il nome `char` deriva dal fatto che la dimensione deve essere sufficiente a contenere un carattere. Tipicamente, allora, una variabile di tipo `char` contiene valori positivi compresi tra 0 e 255. Questo però è solo l'uso più comune. Il C impone che un `char` erediti le operazioni tipiche di un intero, cioè somma, sottrazione, divisione, moltiplicazione e così via. Allora, anche se raramente usato, non è un errore assegnare un valore negativo a una variabile di tipo `char`:

```
char raro = -1;
```

Il tipo `int`, invece, è un intero la cui dimensione è di solito pari alla *parola* (*word*) della macchina; dunque nella maggioranza dei casi una variabile di tipo `int` occupa quattro byte (32 bit). La dimensione di una variabile di tipo `int`, comunque, cambia da macchina a macchina e l'unica relazione universale è:

```
sizeof(char) <= sizeof(int)
```

Ossia: la dimensione di un `char` è minore o uguale alla dimensione di un `int`.
Il tipo `float` corrisponde a un dato numerico di singola precisione in virgola mobile. Anche la dimensione di un `float` dipende dall'architettura della macchina in esame: spesso un `float` occupa 4 byte (32 bit).
Il tipo `double` corrisponde a un dato numerico in virgola mobile, ma in doppia precisione. Anche la dimensione del `double` dipende dalla macchina: in molti casi un `double` occupa 8 byte (64 bit). L'unica relazione universale è:

```
sizeof(float) <= sizeof(double)
```

I quattro tipi fondamentali possono essere ulteriormente specificati per mezzo di due tipi di qualificatori, i *qualificatori di dimensione*

`short` e `long`

e il *qualificatore aritmetico*

```
unsigned
```

I qualificatori di dimensione si applicano al tipo fondamentale `int`:

```
short int
```

definisce un dato numerico la cui dimensione è minore o uguale a quella di `int`:

```
sizeof(short int) <= sizeof(int)
```

mentre

```
long int
```

definisce un dato numerico la cui dimensione è maggiore o uguale a quella di `int`.
Il C standard ANSI prevede anche un tipo `long double`, che definisce un dato numerico la cui dimensione è maggiore o uguale a quella di `double`.
Riassumendo si ha:

```
sizeof(char) <= sizeof(short int) <=sizeof(int) <=sizeof(long int)
```

```
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

Il qualificatore aritmetico si applica invece ai tipi `char` e `int`:

```
unsigned char
unsigned int
```

In entrambi i casi, con `unsigned` si fa uso dell'aritmetica senza segno. Questo significa che i valori numerici contenuti nelle variabili di tipo `unsigned char` e `unsigned int` sono sempre considerati come interi positivi. Nell'uso dei qualificatori la sintassi del C ammette delle abbreviazioni:

```
short            <---->  short int
long             <---->  long int
unsigned         <---->  unsigned int
unsigned short   <---->  unsigned short int
unsigned long    <---->  unsigned long int
```

Il tipo `long double` non può invece essere abbreviato.
Abbiamo visto come la dimensione dei tipi dipenda dalle caratteristiche del processore della macchina ospite. Nella tabella seguente si riportano i dati che si ritrovano in alcune delle più diffuse architetture.

Tipo	Dimensione
char	1 byte
short	2 byte
int	4 byte
long	4 byte
float	4 byte
double	8 byte
long double	8 byte

Con il semplice programma illustrato nel Listato 11.1 si può verificare su qualunque macchina la dimensione dei tipi fondamentali.

```
#include <stdio.h>

main()
{
   int ch, in, sh, lo, fl, dd, ld;

   ch = sizeof(char);
   in = sizeof(int);
   sh = sizeof(short);
   lo = sizeof(long);
   fl = sizeof(float);
   dd = sizeof(double);
   ld = sizeof(long double);

   printf("La dimensione di un char  è      %d\n", ch);
   printf("La dimensione di uno short è      %d\n", sh);
   printf("La dimensione di un int è        %d\n", in);
   printf("La dimensione di un long è       %d\n", lo);
   printf("La dimensione di un float è       %d\n", fl);
   printf("La dimensione di un double è      %d\n", dd);
   printf("La dimensione di un long double è %d\n", ld);

}
```

Listato 11.1 Visualizzazione della dimensione dei tipi fondamentali

11.3 Costanti

Si dice costante un valore che non può essere variato durante l'esecuzione di un programma. A una costante può essere associato un nome simbolico come sua rappresentazione.
Esistono tre principali tipi di costanti: carattere, intere, e floating point; un ulteriore tipo è rappresentato dalle stringhe di caratteri, che vengono trattate come costanti di tipo `char[]`. Ecco quindi alcuni esempi di costante:

```
int                 43,452,-6573
long           -433L, 0L,547343771
double              32.45,-34.4,-0.33e-3,54.3e4
char           'f', 'G', 't', '#'
"stringa"      "Bobo","Frog","ranocchio"
```

In C sono previste quattro categorie di costanti di tipo intero:

- • le costanti decimali;
- • le costanti ottali;
- • le costanti esadecimali;
- • le costanti carattere.

Le costanti decimali sono quelle più comunemente usate e sono della forma:

```
0       1234       976        12345678901234567890
```

Il tipo di una costante decimale è `int` se `int` è sufficiente a contenerla, altrimenti è `long`. Il compilatore avverte se si stanno trattando costanti più grandi di quelli rappresentabili sulla macchina. Per facilitare la programmazione di basso livello il linguaggio consente la definizione di costanti numeriche in sintassi ottale ed esadecimale (purtroppo manca la sintassi binaria):

- • ottale aggiungere una cifra `0` prima della costante
- • esadecimale aggiungere `0x` o `0X` prima della costante

Per esempio: `012,077,-05` sono costanti ottali, mentre `0xAA`, `0xddL`, `0xF` sono costanti esadecimali.
Una costante floating point è di tipo `double`. Il compilatore, in genere, avverte anche in questo caso se si stanno trattando costanti più grandi di quelli rappresentabili sulla macchina. Esempi di costanti floating point sono:

```
1.23     .23      0.23     1.    1.0      1.2e10   1.23e-5
```

Non si possono inserire spazi nel mezzo di una costante floating point. Per esempio, la sequenza di caratteri

```
65.43 e - 21
```

non è una costante floating point ma rappresenta quattro simboli diversi: `65.43`, `e`, `-`, `21`, che provocheranno un errore sintattico.
In C non esiste propriamente un tipo carattere, ma piuttosto un tipo intero che può contenere caratteri. Esiste una speciale convenzione per indicare le costanti di tipo carattere: esse vengono racchiuse tra apici:

```
'a'      '0'      'A'
```

Tali costanti carattere sono realmente delle rappresentazioni simboliche, che corrispondono ai valori interi associati a ogni carattere nell'insieme dei caratteri della macchina. Se il sistema usa la codifica ASCII il valore di `'0'` è 48, mentre se usa la codifica EBCDIC il suo valore è 240.
È possibile definire costanti carattere non limitate ai caratteri alfanumerici ma estese a tutte le 256 combinazioni ottenibili con gli 8 bit di un byte. La sintassi di definizione è:

```
\ooo
```

dove `ooo` rappresenta una maschera di bit interpretata secondo le convenzioni dell'aritmetica ottale. Così, per esempio, si hanno le seguenti corrispondenze.

Maschera ottale	Rappresentazione binaria	Rappresentazione decimale
`\201`	`01000001`	`65`
`\012`	`00001010`	`10`
`\11`	`00001001`	`9`
`\0`	`00000000`	`0`
`\377`	`11111111`	`254`

Se consideriamo la prima linea, l'assegnamento:

```
lettera = '\201';
```

nel codice ASCII corrisponde a:

```
lettera = 'A';
```

Questa rappresentazione è comoda quando si devono utilizzare combinazioni non alfanumeriche, come quelle presenti nelle linee della tabella successive alla prima. Si provi a verificarne la rappresentazione nel codice ASCII.
Per i compilatori conformi allo standard ANSI è prevista anche la codifica esadecimale per le costanti `char`:

```
\xhhh
```

che si ottiene facendo precedere il valore da `\x`: `'\xfa'`, `'\1a'`. Comunque, poiché esistono molte costanti carattere non alfanumeriche di uso comune, il C aiuta ulteriormente il programmatore nella loro definizione.

`\'`	Apice singolo
`\"`	Doppio apice
`\?`	Punto interrogativo
`\\`	Backslash – carattere \
`\a`	"Bell"
`\b`	"Backspace" (^H)
`\f`	"Form feed" (^L)
`\n`	New line (^J)
`\r`	Carriage return (^M)
`\t`	"Tab" (^I)
`\v`	Tabulazione verticale (^V)

Si ricorda che, nonostante l'apparenza, questi sono tutti caratteri singoli.
Le costanti cui si fa riferimento con un nome (o simbolo) sono dette costanti *simboliche*. Una costante simbolica è quindi un nome il cui valore non può essere ridefinito nell'ambito di validità del nome stesso. In C esistono tre tipi di costanti simboliche:

1. 1. costanti rese simboliche per mezzo della parola chiave *`const`*;
2. 2. una collezione di costanti elencato da una enumerazione;
3. 3. costanti simboliche corrispondenti ai nomi di array e funzioni.

La parola chiave `const` può essere aggiunta alla dichiarazione di un oggetto in modo da rendere quell'oggetto una costante invece di una variabile:

```
const int modello = 145;
const int v[] = {1, 2, 3, 4};
```

Il lettore osservi come, non potendosi effettuare assegnazioni a una costante nel corso di un programma, si debba necessariamente procedere alla inizializzazione in fase di dichiarazione:

```
modello = 165;       /* errore */
modello++;           /* errore */
```

Dichiarando un nome `const` si fa in modo che il suo valore non cambi nell'ambito di validità del nome. Dunque `const` è un modificatore di tipo, nel senso che limita i modi in cui un oggetto può essere usato, senza di per sé specificare il tipo di quell'oggetto.
Non è richiesta alcuna memoria dati per allocare una costante, per il semplice motivo che il compilatore conosce il suo valore e lo sostituisce all'interno del programma prima di generare l'eseguibile. L'inizializzatore di un nome di tipo `const` può essere anche una espressione di tipo costante che comunque viene valutata durante la compilazione. Il tipo `const` è utilizzato anche per ottenere costanti in virgola mobile di tipo `float`, non disponibili di per sé:

```
const float pi8 = 3.14159265;
```

La dichiarazione `const int` può essere abbreviata da `const`.
Esiste un metodo alternativo di definire costanti intere, spesso più conveniente dell'uso di `const`. Per esempio:

```
enum { QUI, QUO, QUA };
```

definisce tre costanti intere dette enumeratori alle quali assegna implicitamente dei valori interi sequenziali e crescenti a partire da zero; `enum { QUI, QUO, QUA }` è equivalente a:

```
const QUI = 0;
const QUO = 1;
const QUA = 2;
```

A una enumerazione può essere associato un nome:

```
enum papero { QUI, QUO, QUA };
```

che non è un nuovo tipo ma un sinonimo di `int`. Agli enumeratori possono essere anche assegnati esplicitamente valori:

```
enum valore_simbolo {
       NOME, NUMERO,  FINE,
       PIU = '+', MENO = '-', PER = '*', DIV = '/'
     };
```

La dichiarazione

```
valore_simbolo x;
```

costituisce un utile suggerimento sia per il lettore sia per il compilatore!

11.4 Il trattamento dei bit

Per scrivere applicazioni che controllano dispositivi hardware è necessario disporre di operatori in grado di lavorare sui singoli bit di un registro. Se per esempio si ha una centralina di controllo di un impianto di illuminazione, dove un registro di 16 bit comanda lo stato di 16 lampade, mettere a 1 un bit del registro significa accendere una lampada se questa è spenta, e metterlo a 0 significa spegnerla se questa è accesa. Il C fornisce un ricco insieme di operatori per il trattamento dei bit:

`&`	AND bit a bit
`|`	OR bit a bit
`^`	OR esclusivo
`<<`	shift sinistra
`>>`	shift destra
`~`	complemento a 1

Gli operatori per il trattamento dei bit – si veda più avanti in Figura 11.1 la tavola di priorità complessiva – sono gli stessi disponibili in tutti i linguaggi assembler, e rendono il C quello che alcuni hanno definito "il linguaggio di più alto livello tra quelli a basso livello".

Analizziamo ora uno per uno gli operatori servendoci di esempi. Come premessa occorre precisare che questi operatori non lavorano su variabili di tipo `float` o `double`, ma su variabili di tipo `char` e `int`. In effetti, pensando agli elementi di memoria di un qualsiasi dispositivo hardware (una centralina, una scheda di rete, il controller di un disco) i valori in virgola mobile non hanno alcun significato: i bit sono bit di comando e le uniche operazioni che servono sono quelle di abilitazione/disabilitazione. In altre parole, sui bit di un registro non c'è bisogno di compiere operazioni aritmetiche: i registri sono semplicemente una collezione di bit cui si attribuisce un preciso significato. Se ne deduce, allora, che le operazioni per il trattamento dei bit, con l'eccezione delle operazioni di shift e complemento, si applicheranno tipicamente a variabili di tipo `unsigned char` e `unsigned int`, poiché il segno aritmetico non ha significato.

L'istruzione di AND bit a bit è usata per *spegnere* uno o più bit di una variabile. Infatti, nel confronto bit a bit dell'operazione booleana AND basta avere uno dei due operandi a 0 per produrre 0.

b1	b2	b1 & b2
0	0	0
1	0	0
0	1	0
1	1	1

Se per esempio volessimo mettere a 0 i quattro bit meno significativi di una variabile `x` di tipo `unsigned char`, e lasciare inalterato la stato di quattro bit più significativi, basterebbe mettere in AND `x` con la sequenza:

```
1 1 1 1 0 0 0 0
```

il cui valore ottale è 360. In pratica si tratta di formare una sequenza di bit dove sia posto il valore 1 in corrispondenza dei bit di `x` che devono rimanere invariati e il valore 0 in corrispondenza dei bit di `x` che devono essere trasformati in zero. Se lo stato iniziale dei bit di `x` fosse

```
1 0 1 0 1 0 1 0
```

dopo l'istruzione

```
x = x & '\360';
```

lo stato dei bit di `x` diventerebbe:

```
1 0 1 0 0 0 0 0
```

L'istruzione di OR bit a bit è usata per *accendere* uno o più bit di una variabile. Infatti, nel confronto bit a bit dell'operazione booleana OR basta avere uno dei due operandi a 1 per produrre 1.

b1	b2	b1 | b2
0	0	0
1	0	1
0	1	1
1	1	1

Se per esempio volessimo mettere a 1 i quattro bit meno significativi di una variabile `unsigned char`, e lasciare inalterato la stato di quattro bit più significativi, basterebbe usare l'istruzione:

```
x = x | '\017';
```

Se prima dell'istruzione lo stato dei bit di `x` fosse

```
1 0 1 0 1 0 1 0
```

corrispondendo la costante `'\017'` alla sequenza

```
0 0 0 0 1 1 1 1
```

dopo l'istruzione

```
x = x | '\017';
```

lo stato dei bit di `x` sarebbe:

```
1 0 1 0 1 1 1 1
```

L'istruzione di OR esclusivo bit a bit è usata per *commutare* da 0 a 1 e viceversa un insieme di bit di una variabile. Infatti, nel confronto bit a bit dell'operazione booleana OR esclusivo se i due bit sono uguali si produce 0, se sono diversi si produce 1.

b1	b2	b1 ^ b2
0	0	0
1	0	1
0	1	1
1	1	0

Se per esempio volessimo scambiare 0 con 1 e 1 con 0 nei primi quattro bit più significativi di una variabile `unsigned char`, e lasciare inalterato lo stato dei quattro bit meno significativi, basterebbe usare l'istruzione:

```
x = x ^ '\360';
```

Se prima dell'istruzione lo stato dei bit di `x` fosse

```
1 0 1 0 1 0 1 0
```

corrispondendo la costante `'\360'` alla sequenza

```
1 1 1 1 0 0 0 0
```

dopo l'istruzione

```
x = x ^ '\360';
```

lo stato dei bit di x sarebbe:

```
0 1 0 1 1 0 1 0
```

Si osservi che effettuare l'OR esclusivo di una variabile con se stessa ha l'effetto di mettere a 0 la variabile. L'istruzione

```
x = x ^ x;
```

trasforma sempre `x` in:

```
0 0 0 0 0 0 0 0
```

Il lettore potrebbe obiettare che lo stesso risultato si sarebbe potuto ottenere con l'assegnazione:

```
x = 0;
```

Il fatto è che la pratica di azzerare una variabile facendo l'OR esclusivo con se stessa deriva dall'assembler, dove trova frequente applicazione. Se poi si vogliono scambiare tutti i bit di una variabile da 0 a 1 e viceversa, basta effettuare il complemento a 1 della variabile stessa:

```
x = ~x;
```

Se prima dell'istruzione lo stato dei bit di `x` fosse

```
1 0 1 0 1 0 1 0
```

dopo l'istruzione

```
x = ~x;
```

lo stato dei bit di `x` sarebbe:

```
0 1 0 1 0 1 0 1
```

Le operazioni di shift `<<` e `>>` traslano il primo operando, rispettivamente a sinistra e a destra, di un numero di posizioni corrispondenti al valore del secondo operando. L'effetto delle operazioni di shift cambia a seconda che le si applichi a variabili senza segno o con segno; per quelle con segno il risultato può cambiare a seconda del tipo di macchina. In generale valgono le seguenti regole:

- • se si esegue lo shift a destra di una variabile `unsigned` i bit vacanti a sinistra sono rimpiazzati con degli 0;
- • se si esegue lo shift a destra di una variabile con segno i bit vacanti a sinistra sono rimpiazzati con il bit del segno su certe macchine (shift aritmetico) e con 0 su altre (shift logico);
- • se si esegue lo shift a sinistra di una variabile `unsigned`, i bit vacanti a destra sono rimpiazzati con degli 0;
- • se si esegue lo shift a sinistra di una variabile con segno, i bit vacanti a destra sono rimpiazzati con degli 0 e il bit del segno rimane inalterato.

Vediamo ora alcuni esempi di operazioni di shift.

1)
```
char c;
c = c << 1;
```

Se prima dello shift i bit di `c` fossero stati

```
1 0 0 0 1 0 0 1
```

dopo lo shift avremmo avuto:

```
1 0 0 1 0 0 1 0
```

Avremmo cioè mantenuto a 1 il primo bit di segno.

2)
```
char c;
c = c >> 1;
```

Se prima dello shift i bit di `c` fossero stati

```
1 0 0 0 1 0 0 1
```

dopo lo shift avremmo avuto

```
1 1 0 0 0 1 0 0
```

Il bit vacante a sinistra sarebbe cioè rimpiazzato dal bit del segno.

3)
```
unsigned char c;
c = c >> 1;
```

Se prima dello shift i bit di `c` fossero stati

```
1 0 0 0 1 0 0 1
```

dopo lo shift avremmo avuto

```
0 1 0 0 0 1 0 0
```

Il bit vacante a sinistra sarebbe cioè rimpiazzato dal bit 0.

Si noti come nello shift a sinistra i bit vacanti di destra siano sempre rimpiazzati da 0. Ciò equivale a moltiplicare per multipli di 2. Così, per esempio, l'istruzione:

```
x = x << 2;
```

produce su `x` l'effetto di una moltiplicazione per 4.
Le operazioni di shift sono spesso usate congiuntamente alle operazioni booleane bit a bit per controllare lo stato di uno o più bit di una variabile. Per sapere, per esempio, se il quarto bit a partire da destra della variabile `c` è 0 oppure 1 potremmo procedere così:

```
if( (x >> 3) & 01 )
   printf("In quarto bit di x è 1");
else
   printf("In quarto bit di x è 0");
```

Nell'esempio abbiamo raccolto `x>>3` all'interno di parentesi tonde: infatti avendo l'operatore `>>` minore priorità di `&`, in caso contrario sarebbe stato effettuato all'interno dell'espressione prima l'AND bit a bit tra 3 e 01 e poi lo shift a destra di `x`.

È il momento di ampliare la tavola gerarchica delle priorità degli operatori (Figura 11.1), dove inseriamo anche la trasformazione di tipo indicata nella tabella con `(tipo)`, che studieremo approfonditamente nel prossimo paragrafo. Si tenga conto che per gli operatori binari per il trattamento dei bit, come per quelli aritmetici, è possibile utilizzare la notazione:

```
variabile [operatore]= espressione
```

quando una variabile appare sia a sinistra sia a destra di un operatore di assegnamento. Pertanto l'istruzione:

```
x = x << 2;
```

può essere scritta come

```
x <<= 2;
```

```
!   -   ++   --   sizeof   (tipo)
*   /   %
+   -
>>   <<
>   >=   <   <=
==   !=
&
^
|
&&
||
?:
=   +=   -=   *=   /=   %=   &=   ^=   |=   <<=   <<=
,
```

Figura 11.1 Tavola di priorità degli operatori

11.5 Le conversioni di tipo

I valori di una variabile possono essere convertiti da un tipo a un altro. Questa conversione può avvenire implicitamente o può essere comandata esplicitamente. In ogni caso il programmatore deve sempre tener conto degli eventuali effetti delle conversioni implicite di tipo; quelle ammesse sono le seguenti:

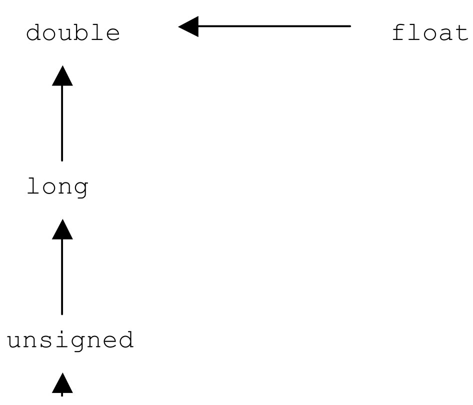
1) 1) il tipo `char` è implicitamente convertibile in `int`, `short int` e `long int`. La conversione di un carattere in un valore più lungo può riguardare o meno l'estensione del segno, a seconda del tipo di macchina;
2) 2) il tipo `int` è implicitamente convertibile in `char`, `short int`, `long int`. La conversione a un intero più lungo produce l'estensione del segno; la conversione a un intero più corto provoca il troncamento dei bit più significativi. Il tipo `int` è implicitamente convertibile anche in `float`, `double` e `unsigned int`. Per quest'ultimo caso la sequenza dei bit dell'`int` è interpretata come `unsigned`;
3) 3) i tipi `short int` e `long int` possono essere convertiti implicitamente come `int` (punto 2);
4) 4) il tipo `float` può essere convertito in `double` senza alcun problema. Il tipo `float` può essere anche convertito in `int`, `short int` e `long int`. Il risultato non è definito quando il valore `float` da convertire è troppo grande, perché cambia da macchina a macchina;
5) 5) il tipo `double` può essere convertito in `float`, potendo provocare troncamento nell'arrotondamento. Inoltre, per il `double` vale quanto detto per il `float` a proposito di `int`, `short int` e `long int`.

> ✓ **NOTA**
> Le conversioni implicite di tipo vengono effettuate per rendere conformi, se possibile, i tipi di due operandi in un'espressione e i tipi dei parametri attuali e formali nel passaggio dei parametri di una funzione. In C sono possibili molte conversioni implicite di tipo, ma ci limiteremo a considerare solo quelle che il programmatore può ragionevolmente ammettere. Per tutte le altre conversioni, non è saggio affidarsi al compilatore, soprattutto per motivi di leggibilità e portabilità delle applicazioni, e, pertanto, si raccomanda di usare la conversione esplicita di tipo.

In C l'esecuzione degli operatori aritmetici effettua la conversione implicita degli operandi se questi non sono omogenei. Lo schema di conversione usato è comunemente detto *conversione aritmetica normale*, ed è descritto dalle regole seguenti.

- • Convertire gli operandi `char` e `short int` in `int`; convertire gli operandi `float` in `double`.
- • Se uno degli operandi è di tipo `float`, allora l'altro operando, se non è già `double`, è convertito in `double`, e il risultato è `double`.
- • Se uno degli operandi è di tipo `long`, allora l'altro operando, se non è già `long`, viene convertito in `long`, e il risultato è `long`.
- • Se un operando è di tipo `unsigned`, allora l'altro operando, se non è già `unsigned`, è convertito in `unsigned`, e il risultato è `unsigned`.
- • Se non siamo nella situazione descritta dai casi 2 e 4, allora entrambi gli operandi debbono essere di tipo `int` e il risultato è `int`.

In forma più sintetica, queste regole di conversione possono essere illustrate dal diagramma seguente.



Si osservi dal diagramma come le conversioni orizzontali vengano sempre effettuate, mentre quelle verticali avvengano solo se necessario. Nel frammento di codice

```
short i;
...
i = i + 4;
```

si ha che la variabile `i` di tipo `short` è convertita in `int` e sommata alla costante intera 4. Il risultato della somma è convertito in `short`, prima dell'assegnazione. In quest'altro esempio:

```
double dd;
int    i;
...
i = i + dd;
```

la variabile intera `i` è convertita in `double`, sommata alla variabile `double dd` e il risultato della somma, prima dell'assegnazione, è convertito in `int`, con eventuale perdita di precisione.
Un'espressione di un certo tipo può essere esplicitamente convertita in un altro tipo per mezzo dell'operazione detta di *cast*. Abbiamo già incontrato il cast nel Capitolo 9, a proposito delle funzioni di allocazione e deallocazione dinamica della memoria `malloc`, `calloc` e `free`, e dell'indirizzamento assoluto della memoria. La sintassi generale del cast è:

```
(nome_tipo) espressione
```

Il valore di *`espressione`* è trasformato in un valore il cui tipo è *`nome_tipo`*. Esempi di *`nome_tipo`* sono:

```
char
char[8]
char *
char ()
int
void
```

In pratica *`nome_tipo`* è lo stesso che si usa nella dichiarazione delle variabili. Si consideri il semplice esempio:

```
char *pc;
int   a;
...
a = 0177777;
pc = (char *)a;
```

Alla variabile `a` di tipo `int` viene assegnata la costante ottale di tipo `int 0177777`. Successivamente questo valore contenuto in `a` è esplicitamente convertito nel tipo puntatore a `char` e assegnato alla variabile di tipo `char *`, `pc`. Si faccia bene attenzione: il tipo della variabile `a`, e così il suo contenuto, rimane `int` anche dopo il cast. Semplicemente il compilatore valuta l'espressione `a`, e il risultato di tale espressione è temporaneamente convertito in `char *`, e così convertito è assegnato alla variabile `pc`.
I cast sono principalmente usati per:

1. 1. convertire puntatori di tipo diverso ;
2. 2. eseguire calcoli con la precisione desiderata senza dover introdurre una variabile temporanea.

Finora abbiamo sempre usato il cast per operare conversioni tra puntatori che puntano a tipi diversi, ma può essere utile usarlo anche nelle espressioni aritmetiche:

```
int i;
...
```

```
i = ((long)i * 10) /9;
```

Trasformare la variabile `int i` in `long`, prima della moltiplicazione, garantisce che le operazioni di moltiplicazione e divisione vengano eseguite con una precisione maggiore. Nell'esempio:

```
int i, k;
...
k = (i+7)*3 + (double)(k+2*i);
```

la moltiplicazione tra `(i+7)*3` è una moltiplicazione tra `int` ed è eseguita per prima, mentre il risultato dell'espressione `(k+2*i)` è convertito in `double`, e la somma tra le due espressioni `(i+7)*3 e (k+2*i)` è di tipo `double`. Il risultato della somma è poi implicitamente convertito in `int`, con eventuale perdita di precisione.

11.6 Funzione di visualizzazione

Abbiamo ripetutamente utilizzato negli esempi la funzione di immissione dati, `scanf`, e quella di visualizzazione, `printf`. La sintassi di queste funzioni è stata usata solo sommariamente, ed è ora giunto il momento di definirla (in questo paragrafo e nel prossimo) in modo più completo e generale.
La sintassi della funzione `printf` è:

```
int printf(char *format, arg1, arg2, ...)
```

La funzione `printf` converte, ordina in un formato e stampa sul video (detto standard output) un numero variabile di argomenti *`arg1`*, *`arg2`*, ... sotto il controllo della stringa `format`. Questa stringa dettaglia due diverse informazioni: un insieme di caratteri ordinari, che vengono direttamente inviati sul video, e una specifica di conversione per ognuno degli argomenti da visualizzare. Ogni specifica di conversione inizia con il carattere `%` e termina con il carattere di conversione. Dopo il carattere `%` possono essere presenti i simboli illustrati nei punti successivi.

1) 1) Zero, uno o più *flag* che modificano il significato della specifica di conversione. Tipici flag per la modifica delle conversioni sono i segni meno, più e il carattere vuoto:

`-`	il risultato della conversione sarà accostato a sinistra nel campo;
`+`	il risultato di una conversione con segno inizierà sempre con il segno (+ o –);
`carattere vuoto`	se il primo carattere di una conversione con segno non è un segno, un carattere vuoto precede la visualizzazione del risultato. Ciò significa che se sono specificati sia il carattere vuoto sia il +, il carattere vuoto sarà semplicemente ignorato.

2) 2) Una stringa opzionale di cifre decimali che specifica l'ampiezza minima del campo riservato al corrispondente argomento. Se il valore convertito ha meno caratteri di quelli specificati nell'ampiezza del campo, il valore è accostato a sinistra o a destra del campo, dipendentemente dal flag di accostamento. Se il flag di accostamento non è presente il valore viene posto a destra del campo, se è – (vedi punto precedente) il valore viene posto a sinistra.
3) 3) Un punto che separa l'ampiezza di campo dalla successiva stringa di cifre, detta precisione.
4) 4) La *precisione* che stabilisce:
   - – il numero minimo di cifre che debbono apparire per le conversioni `d, o, u, x` e `X`;
   - – il numero di cifre decimali dopo il punto decimale per le conversioni `e` e `f`;
   - – il massimo numero di cifre significative per la conversione `g`;
   - – il massimo numero di caratteri che debbono essere visualizzati in una stringa relativamente alla conversione `s`.

   La precisione prende la forma di un punto seguito da una stringa di cifre: una stringa nulla è trattata come zero.
6) 1) Una lettera opzionale `l` che specifica l'applicazione delle conversioni `d, o, u, x`, e `X` a un argomento di tipo `long int`.
7) 2) Il carattere di conversione da applicare all'argomento corrispondente. I caratteri di conversione sono:

`d`	l'argomento è convertito in notazione decimale;
`o`	l'argomento è convertito in notazione ottale senza segno e senza lo zero iniziale;
`x`	l'argomento è convertito in notazione esadecimale senza segno e senza lo `0x` iniziale;
`u`	l'argomento è convertito in notazione decimale senza segno;
`c`	l'argomento è considerato un carattere singolo;

s l'argomento è una stringa; tutti i caratteri che precedono la prima occorrenza del carattere '\0' sono visualizzati a meno che non si superi il numero di caratteri specificati nella precisione;

e l'argomento è interpretato come un float o un double ed è convertito in notazione decimale secondo il formato [-]m.nnnnnnE[+/-]xx dove la lunghezza della stringa di n è specificata dalla precisione. Il valore di default della precisione è 6;

f l'argomento è interpretato come float o double ed è convertito in notazione decimale secondo il formato [-]mmm.nnnnn dove la lunghezza della stringa di n è specificata dalla precisione;

g permette automaticamente di scegliere tra %e e %f a seconda di quale sia quello più corto; gli zero non significativi non sono visualizzati.

Si osserva che se il carattere che segue il simbolo % non è uno dei caratteri di conversione, allora esso è visualizzato. Quindi per stampare il carattere di percentuale basta specificare %%. Il primo argomento di printf è quello che dice alla funzione, interpretando i caratteri di conversione, quanti argomenti printf dovrà trattare e il rispettivo tipo. Se il programmatore non fa in modo che ci sia coerenza tra il numero e il tipo di caratteri di conversione e gli argomenti *arg1*, *arg2*, ... la funzione printf produrrà i più bizzarri effetti. La printf ritorna il numero di caratteri che ha visualizzato. Alcuni esempi di specifica relativi a una stringa sono:

```
%10s      |Ciao, lettore|
%-10s     |Ciao, lettore|
%20s      |       Ciao, lettore|
%-20s     |Ciao, lettore       |
%20.10s   |          Ciao, lett|
%-20.10   |Ciao, lett          |
%.10s     |Ciao, lett|
```

dove la dimensione del campo è stata convenzionalmente delimitata con il simbolo |.

11.7 Funzione di immissione

La sintassi di scanf è:

```
int scanf(char *format, puntat1, puntat2, ...)
```

La funzione scanf legge un insieme di caratteri da tastiera (detta standard input), li interpreta secondo il formato specificato dalla stringa format e li memorizza negli argomenti puntati da *puntat1*, *puntat2* ecc. La stringa di formato, detta anche controllo, contiene le specifiche di conversione, che sono usate da scanf per interpretare la sequenza di immissione. Nel controllo possono essere presenti i simboli riportati nei punti successivi:

- • caratteri vuoti (blank, tab, newline, formfeed), che sono ignorati;
- • caratteri normali escluso %, che dovrebbero coincidere con il prossimo carattere non vuoto della sequenza di ingresso;
- • le specifiche di conversione formate dal carattere %, da un carattere * opzionale di soppressione assegnamento, un'ampiezza massima di campo opzionale, un carattere l o un carattere o opzionali che denotano la dimensione della variabile che ospita il valore in ingresso, e il codice di conversione.

Una specifica di conversione applica la conversione al successivo campo di immissione dati; il risultato della conversione è poi allocato nella variabile puntata dal corrispondente argomento, a meno che non sia presente l'asterisco * di soppressione assegnazione. Con quest'ultimo meccanismo è possibile definire dei campi di immissione che sono semplicemente saltati, poiché non danno luogo ad alcuna assegnazione. Un campo di immissione è definito come una stringa di caratteri non vuoti che si estende non oltre l'ampiezza di campo o fino al prossimo carattere vuoto.
Il codice di conversione stabilisce l'interpretazione del campo di immissione. Codici di conversione ammessi sono:

d ci si aspetta un intero decimale in immissione; il corrispondente argomento deve essere un puntatore a intero;

o ci si aspetta un intero ottale, senza lo zero iniziale; il corrispondente argomento deve essere un puntatore a intero;

x ci si aspetta un intero esadecimale, senza lo 0x iniziale; il corrispondente argomento deve essere un puntatore a intero;

h ci si aspetta un intero short; il corrispondente argomento deve essere un puntatore a intero;

s ci si aspetta una stringa di caratteri; il corrispondente argomento deve essere un puntatore a un array di

caratteri grande abbastanza da contenere i caratteri specificati e il tappo '\0', che è aggiunto automaticamente. Il campo di immissione termina con un carattere vuoto;

`c` ci si aspetta un singolo carattere; il corrispondente argomento deve essere un puntatore a carattere. In questo caso non sono trascurati i caratteri vuoti. Per leggere il prossimo carattere non vuoto occorre usare la specifica `%1s`. Se si specifica una ampiezza di campo, l'argomento corrispondente deve riferirsi a un array di caratteri, per effettuare la lettura del numero di caratteri specificato;

`e,f,g` ci si aspetta un numero in virgola mobile; il corrispondente argomento deve essere un puntatore a `float`. Il formato di immissione dei numeri in virgola mobile è una stringa di cifre, in cui opzionalmente si indica anche il segno, che può contenere un punto decimale, seguito da campo esponente opzionale formato da `E` oppure `e` seguito da un intero, eventualmente con segno.

I caratteri di conversione `d`, `u`, `o` e `x` possono essere preceduti da `l` oppure da `h` per specificare il fatto che il corrispondente argomento è un puntatore a `long` oppure a `short` invece che a `int`. Analogamente, i caratteri `e`, `f` e `g` possono essere preceduti da `l` per indicare che il corrispondente argomento è un puntatore a `double` invece che a `float`.
La conversione effettuata dalla funzione `scanf` sullo standard input termina quando si incontra una costante `EOF`, alla fine della stringa di controllo, o quando si incontra un carattere in immissione che è in contraddizione con la stringa di controllo. In quest'ultimo caso il carattere che ha provocato la contraddizione non viene letto dallo standard input.
La funzione `scanf` ritorna il numero di immissioni che è riuscita a concludere con successo. Questo numero può anche essere zero, nel caso in cui si verifichi immediatamente un conflitto tra un carattere in immissione e la stringa di controllo. Se l'immissione si conclude prima del primo conflitto o della prima conversione, `scanf` restituisce un `EOF`.
Esempi classici di uso di `scanf` sono:

```
int i; float x; char nome[30];
...
scanf("%d%f%s", &i, &x, nome);
```

Sulla linea di immissione si potrebbe avere

```
8 54.52E-1  Abel
```

memorizzando 8 in `i`, 5.452 in `x`, e la stringa `"Abel"` in `nome`. Usando sempre la medesima definizione di variabili

```
scanf("%2d%f%*d %s", &i, &x, &nome);
```

e avendo sulla linea di immissione

```
56789 0123 babbo
```

verrà assegnato 56 a `i`, 789.0 a `x`, verrà ignorato 0123 e assegnato `babbo\0` a `nome`.
Infine si ricordi che l'argomento di `scanf` deve essere sempre un puntatore. Purtroppo è molto facile scordarlo e di programmi in cui si scrivono istruzioni tipo

```
int n;
...
scanf("%d", n));
```

invece di

```
scanf("%d", &n);
```

è pieno il mondo!

11.8 Immissione ed emissione su stringa

Esistono altre due funzioni, `sscanf` e `sprintf`, che eseguono le medesime conversioni di `scanf` e `printf`, ma operando su stringa invece che su standard input e standard output. Il prototype di queste due funzioni è:

```
int sprintf(char *s, char *format, arg1, arg2, ...)
int sscanf(char *s, char *format, punt1, punt2, ...)
```

La funzione `sprintf` è comunemente usata per scrivere in un determinato formato variabili C dentro una stringa. La stringa in cui scrivere gli argomenti *arg1*, *arg2*, `...`, secondo il formato specificato dalla stringa `format`, è il primo argomento della funzione, `s`. Per ciò che riguarda la specifica di formato valgono le medesime convenzioni di `printf`. Tipicamente `sprintf` è usata per preparare delle stringhe che saranno poi usate da altri programmi di visualizzazione più semplici che, per esempio, non utilizzano alcun meccanismo di formattazione dei dati. A titolo illustrativo si consideri il programma del Listato 11.2, nel quale il contenuto delle variabili `i` e `d` è copiato nella stringa `bufusc` che viene successivamente visualizzata da `printf`.

```
#include <stdio.h>

int    i = 80;
double d = 3.14546;
main()
{
   int  numusc;
   char bufusc[81];

   numusc = sprintf(bufusc, "Il valore di i = %d e \
   il valore di d = %g\n", i, d);

   printf("sprintf() ha scritto %d caratteri e il \
   buffer contiene:\n%s", numusc, bufusc);
}
```

Listato 11.2 Copia su stringa

La funzione `sscanf` ha lo scopo di leggere dei caratteri da una stringa, `s`, che possiamo chiamare buffer di caratteri, convertirli e memorizzarli nelle locazioni puntate da *punt1*, *punt2*, ... secondo il formato specificato dalla stringa `format`. Per ciò che riguarda la specifica di formato valgono le medesime convenzioni di `scanf`. Tipicamente `sscanf` è usata per la conversione di caratteri in valori. Generalmente si leggono le stringhe da decodificare, per esempio, con la funzione `gets`, e poi si estraggono dei valori dalla stringa con `sscanf`.
La funzione `sscanf` restituisce il numero di campi che sono stati letti, convertiti e assegnati a variabili con successo. Se la stringa dovesse finire prima della fine dell'operazione di lettura, `sscanf` ritornerebbe il valore costante `EOF`, definito in `stdio.h`. A titolo di esempio si consideri il programma del Listato 11.3, che accetta in ingresso assegnazioni di variabili secondo il formato "*nome* = <*valore*>". La funzione `gets` brutalmente legge tutta la stringa corrispondente all'assegnazione e la funzione `sscanf` la scompone nei due elementi `nome` e `valore` del nome.

```
#include <stdio.h>

main()
{
   double valore;
   char buf[31], nome[31];

   printf("Inserire una variabile nel formato\
   \"nome = <valore>\":");
   gets(buf);

   /* Con sscanf() si separano il nome dal valore */

   sscanf(buf, " %[^=] = %lf", nome, &valore);

   printf("La variabile %s vale %f\n", nome, valore);
```

```
}
```

Listato 11.3 Esempio di uso `sscanf`

11.9 Esercizi

1. Verificare con un apposito programma quale dei seguenti valori assume la variabile `i` di tipo `short int` se durante l'esecuzione di una `scanf("%d", &i)` vengono immessi dall'utente i valori 32768 e 32769. E se `i` è di tipo `int`? Le risposte date sono assolute o possono variare, e perché?

2. Verificare con un apposito programma quale dei seguenti valori assume la variabile `c` di tipo `char` se durante l'esecuzione di una `scanf("%d", &c)` vengono immessi dall'utente i valori 300: 300, 44 o 6. Sapreste motivare il risultato? Quale valore assume `c` se vengono immessi 257, 256 o 255?

3. Dato il seguente frammento di programma

```
unsigned char c;
c = '\166' & '\360';
printf("c: %o\n", c);
```

che cosa visualizzerà la `printf`, e perché? E se al posto di `'\166'` avessimo scritto `'\100'` oppure `'\0'` oppure `'\111'` o `'\110'`?

4. Dato il seguente frammento di programma

```
unsigned char c;
c = '\111' ^ '\360';
printf("c: %o\n", c);
```

che cosa visualizzerà la `printf`, e perché? E se al posto di `'\111'` avessimo scritto `'\321'` o `'\350'`?

5. Dato il seguente frammento di programma

```
unsigned char c;
c = ~'\351';
printf("c: %o\n", c);
```

che cosa visualizzerà la `printf`? E se al posto di `'\351'` avessimo scritto `'\222'` o `'\123'`?

5. Se la variabile `c` di tipo `unsigned char` ha valore `'\123'`, quali valori stamperebbe l'istruzione `printf("c: %o\n", c)` dopo ognuno dei seguenti singoli assegnamenti?

```
c = c >> 1;
c = c >> 2;
c = c >> 3;
c = c << 1;
c = c << 2;
c = c << 3;
```

6. Se la variabile `c` è di tipo `unsigned char`, quali valori stampano le due `printf` del seguente frammento di programma?

```
c = '\321';
c = c >> 1;
printf("c: %o\n", c);
c = '\321';
c = c << 1;
printf("c: %o\n", c);
```

12.1 Definizione di tipo derivato

A partire dai tipi fondamentali è possibile costruire nuove classi, dette *tipi derivati*. Abbiamo già incontrato alcuni dei principali tipi derivati. Gli array, le funzioni e i puntatori sono esempi di tipi derivati, nel senso che per essere specificati hanno bisogno di riferirsi a un tipo base. Se i tipi fondamentali rappresentano i più elementari costrutti trattati da un calcolatore, i tipi derivati possono essere usati per modellare oggetti più complessi e più vicini alle strutture del mondo reale. La capacità di poter derivare un tipo da altri è un meccanismo potente che il linguaggio C mette a disposizione del programmatore per risolvere una classe di problemi molto ampia.
Tra i tipi derivati che ancora non abbiamo preso in considerazione vi sono le strutture, le unioni e i campi. Di ognuno di essi indicheremo la sintassi e la modalità d'uso. Tratteremo poi un insieme di tipi derivati che nascono della composizione di altri tipi derivati. In particolare, parleremo di puntatori e funzioni, strutture e funzioni, puntatori e array e puntatori di puntatori.

12.2 Strutture

Per mezzo di un array è possibile individuare mediante un nome e un indice un insieme di elementi dello stesso tipo. Se per esempio volessimo rappresentare un'area di memoria di 200 byte su cui andare a scrivere o a leggere dei dati, potremmo definire una variabile `buf`:

```
int buf[100];
```

Ogni locazione di questa memoria verrebbe individuata attraverso un indice. Così

```
buf[10]
```

rappresenterebbe l'undicesima locazione di memoria del buffer.

Ci sono però problemi in cui è necessario aggregare elementi di tipo diverso per formare una struttura. Per esempio, se si vuol rappresentare il concetto di data basta definire una *struttura* siffatta:

```
struct   data     {
        int   giorno;
        char  *mese;
        int   anno;
      };
```

Si osservi come per effetto di questa dichiarazione si sia introdotto nel programma un nuovo tipo, il tipo `data`. D'ora in poi sarà possibile definire variabili in cui tipo è `data`:

```
struct data oggi;
```

Come deve essere interpretata la variabile `oggi` di tipo `data`? Semplicemente come una variabile strutturata, composta di tre parti: due di tipo `int` – `giorno` e `anno` – e una di tipo `stringa` – `mese` –. La sintassi generale per la definizione di una struttura è:

```
struct nome_struttura  {
         tipo_membro nome_membro1;
         tipo_membro nome_membro2;
         ...
         tipo_membro nome_membroN;
      };
```

Gli elementi della struttura sono detti membri; essi sono identificati da un nome, *`nome_membro`*, e da un tipo, *`tipo_membro`*, che può essere sia fondamentale sia derivato. Nell'esempio della struttura `data` si ha che `giorno` e `anno` sono di tipo `int`, cioè un tipo fondamentale, mentre `mese` è di tipo `char *`, cioè di tipo derivato. Si osservi il punto e virgola posto in coda alla definizione di una struttura: è questo uno dei rari casi in cui in C occorre mettere un "`;`" dopo una parentesi graffa; il lettore presti la dovuta attenzione a questo particolare, spesso trascurato.
Una volta definita una struttura, *`nome_struttura`* diviene un nuovo tipo a tutti gli effetti. Si possono allora definire variabili il cui tipo è *`nome_struttura`*:

```
struct nome_struttura nome_variabile;
```

Per esempio si può scrivere:

```
struct data oggi, ieri, compleanno;
```

dove le variabili `oggi`, `ieri` e `compleanno` sono variabili di tipo `data`. Esiste anche un'altra sintassi per la definizione di variabili di tipo struttura. Per esempio:

```
struct automobile {
        char *marca;
        char *modello;
        int  venduto;
     }  a1, a2;
```

introduce la struttura `automobile`, e al contempo, anche due variabili di tipo `automobile`, `a1` e `a2`. In luogo di questa definizione di struttura e di variabili insieme, avremmo potuto usare la forma equivalente:

```
struct automobile  {
        char *marca;
        char *modello;
        int  venduto;
     };
     struct automobile a1, a2;
```

È lasciato alla sensibilità del lettore stabilire quale delle due convenzioni usare per definire una variabile struttura.

Per poter accedere ai campi di una variabile di tipo struttura si fa uso dell'operatore punto (`.`):

```
oggi.giorno = 25;
     oggi. mese = "Dicembre";
     oggi. anno = 2001;
     ieri.anno = oggi.anno;
```

Con le prime tre istruzioni si assegna la terna di valori `<25,"Dicembre",2001>` alla variabile `oggi` e con la quarta si rendono uguali l'anno di `ieri` con quello di `oggi`.

La sintassi generale con cui si fa riferimento a un membro è

`nome_variabile_struttura.nome_membro`

Per esempio, quando si vuole visualizzare il contenuto di una variabile struttura può essere necessario ricorrere all'operatore punto: si veda il Listato 12.1, la cui esecuzione produrrà il seguente risultato:

```
marca auto  = FERRARI
     modello auto  = F40
     vendute  = 200
     marca auto  = OPEL
     modello auto  = ASTRA
     vendute  = 1200
```

```
/* Esempio di definizione di una struttura */

#include <stdio.h>

struct automobile {
  char *marca;
  char *modello;
  int venduto;
};

main()
{
struct automobile a1, a2;

a1.marca = "FERRARI";
a1.modello = "F40";
a1.venduto = 200;

a2.marca = "OPEL";
a2.modello = "ASTRA";
```

```
a2.venduto = 1200;

printf("marca auto = %s\n", a1.marca);
printf("modello auto = %s\n", a1.modello);
printf("vendute = %d\n", a1.venduto);
printf("marca auto = %s\n", a2.marca);
printf("modello auto = %s\n", a2.modello);
printf("vendute = %d\n", a2.venduto);
}
```

Listato 12.1 La struttura `automobile`

Come si può osservare, le variabili `a1` e `a2` sono visualizzate "stampando" separatamente i membri `marca`, `modello` e `venduto` che le compongono.
Talvolta ci si può riferire a una variabile di tipo struttura nel suo insieme, attraverso il nome, senza dover far riferimento ai singoli membri. Per esempio, in C è possibile effettuare assegnazioni tra variabili struttura dello stesso tipo:

```
struct data compleanno, oggi;
     compleanno = oggi;
```

è una assegnazione consentita che effettua una copia di tutti i valori dei membri di `oggi` nei corrispondenti valori dei membri di `compleanno`. Due strutture corrispondono a due tipi differenti anche se hanno gli stessi membri. Per esempio:

```
struct s1  {
        int a;
     };
     struct s2  {
        int a;
     };
```

sono due tipi struttura differenti:

```
struct s1 bob;
     struct s2 alex;
     ...
     alex = bob; /* errore: tipi dati discordi */
```

Quest'ultima assegnazione darebbe luogo a un errore, poiché si è tentato di assegnare una variabile di tipo `s1` a una di tipo `s2`. Inoltre si ricordi che i tipi struttura sono diversi anche dai tipi fondamentali. Per esempio, in

```
struct s1 bobo;
     int i;
     ...
     bobo = i; /* errore: tipi dati discordi */
```

l'ultima assegnazione provocherebbe un errore a causa della disomogeneità di tipo.
È possibile fare riferimento a una variabile struttura anche attraverso un puntatore, cioè tramite una variabile abilitata a contenere l'indirizzo di una variabile di tipo struttura:

```
struct data *pd, oggi, compleanno;
     pd = &oggi;
     (*pd).giorno = 31;
     (*pd).mese = "Gennaio";
     (*pd).anno = 2001;
```

Attraverso il puntatore `pd` si possono raggiungere i membri della variabile struttura `oggi`. Le parentesi tonde che circoscrivono `*pd` sono necessarie perché l'operatore "`.`" ha priorità maggiore rispetto all'operatore "`*`".
Poiché in C si accede frequentemente a una variabile struttura tramite puntatore per evitare costrutti sintattici laboriosi, è stato introdotto l'operatore freccia `->` per accedere direttamente ai membri di una variabile strutturata puntata da un puntatore. Nel frammento seguente le ultime tre istruzioni sono perfettamente equivalenti alle corrispondenti dell'esempio precedente:

```
struct data *pd, oggi, compleanno;
```

```
     pd = &oggi;
     pd->giorno = 31
     pd->mese = "Gennaio";
     pd->anno = 2001;
```

I puntatori a struttura sono molto usati specialmente per passare una struttura a una funzione, e per far sì che una funzione ritorni un risultato di tipo struttura:

```
int numero_mese(struct data *dt)
     {
        if(dt->mese == "Gennaio")
           return(1);
        else
        if(dt->mese == "Febbraio")
           return(2);
        else
        ...
        if(dt->mese == "Dicembre")
           return(12)
        else
           return(0);
     }
```

Alla funzione `numero_mese` viene passato un puntatore a variabile di tipo `data` e si ottiene il numero del mese relativo alla data puntata dal puntatore. Nel caso in cui il nome del mese non corrisponda ad alcuno di quelli conosciuti, la funzione ritorna un codice di errore pari a 0.
Infine, una ultima considerazione a proposito della inizializzazione di una variabile di tipo struttura. Abbiamo visto come per mezzo dell'operatore punto sia possibile assegnare dei valori a una variabile struttura. In alternativa si può usare una sintassi analoga a quella usata per inizializzare gli array:

```
struct automobile {
        char *marca;
        char *modello;
        int venduto;
     };
     struct automobile a = {"FERRARI", "F40", 200};
```

oppure

```
struct data {
        int giorno;
        char *mese;
        int anno;
     } oggi = {25, "Dicembre", 2001};
```

Questo tipo di inizializzazione è ammesso solo se le corrispondenti variabili sono di tipo globale, cioè se sono definite all'esterno di un blocco ▬.

12.3 Unioni

Le *unioni* (`union`) sono analoghe alle strutture: introducono nel programma una nuova definizione di tipo e sono costituite da un insieme di membri, che possono essere – in generale – di tipo e dimensione diversa. I membri di una unione, però – a differenza di una struttura –, coincidono, cioè hanno lo stesso indirizzo e vanno a occupare le medesime locazioni di memoria. Questo implica che l'occupazione di memoria di una unione coincide con quella del membro dell'unione di dimensione maggiore. La sintassi generale di una `union` è analoga a quella delle strutture:

```
union nome_unione {
          tipo_membro nome_membro1;
          tipo_membro nome_membro2;
          ...
          tipo_membro nome_membroN;
```

```
	};
```

Anche le `union` sono nuovi tipi che vengono introdotti in un programma, e le variabili dichiarate di tipo `union` possono, in tempi diversi, assumere oggetti di tipo e dimensione differenti in accordo alle specifiche dei membri dell'unione. Nell'esempio

```
union fantasma {
		int i;
		long d;
		char c;
	};
	union fantasma a;
```

la dimensione della variabile `a` coincide con quella della unione `fantasma` e corrisponde allo spazio occupato da un `long` (in alcune implementazioni 4 byte). Si noti come in base al tipo del membro si abbia una diversa allocazione della variabile `a` (Figura 12.1).

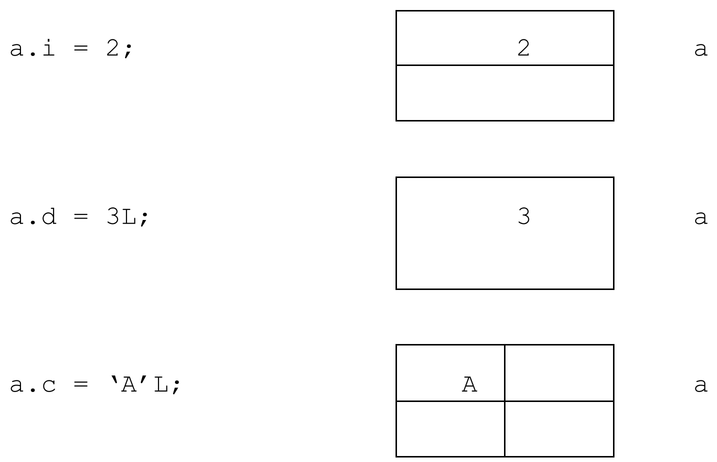


Figura 12.1 Rappresentazione di una unione

Le unioni vengono preferite alle strutture quando si hanno delle variabili che possono assumere più tipi e/o dimensioni diverse a seconda delle circostanze. Per esempio, supponiamo di voler rappresentare per mezzo di una struttura una tavola di simboli, cioè una tabella i cui elementi siano una coppia <nome, valore> e dove il valore associato a un nome può essere o una stringa o un numero intero. Una possibile struttura per la tavola dei simboli potrebbe essere:

```
struct tav_sim {
		char *nome;
		char tipo;
		char *val_str; /* usato se tipo ==  's'  */
		char val_int;  /* usato se tipo ==  'i'  */
	};
	void stampa_voce(struct tav_sim *p)
	{
	   switch (p->tipo) {
	   case  's':
	      printf("%S", p->val_str);
	      break;
	   case  'i':
	      printf("%d", p->val_int);
	      break;
	   default:
	      printf("valore di tipo non corretto\n");
```

```
            break;
        }
    }
```

La funzione `stampa_voce` ha lo scopo di visualizzare la decodifica di una voce della tabella, voce che corrisponde a una variabile di tipo `tav_sim`, passata alla funzione `stampa_voce` tramite un puntatore `p`. Si osserva però che una voce della tavola dei simboli, a seconda dell'attributo tipo, ha una decodifica di tipo stringa o di tipo intero, ma mai entrambe. La struttura che si è proposta è allora ridondante, cioè comporta un'occupazione di memoria superiore a quella richiesta. È in situazioni come questa, allora, che si fa ricorso alle unioni. I due attributi `val_str` e `val_int` si trattano come membri di una unione:

```
struct tav_sim {
        char *nome;
        char tipo;
        union {
           char *val_str; /*usato se tipo ==  's'  */
           int  val_int; /* usato se tipo ==  'i'  */
        };
    }
```

Con questa soluzione i programmi che inseriscono, cancellano, ricercano o stampano una voce della tavola dei simboli rimangono invariati, e per mezzo del concetto di unione si ha che i due attributi `val_str` e `val_int` hanno lo stesso indirizzo, ovvero sono allocati nella medesima area di memoria.

Talvolta le unioni sono usate anche per effettuare conversioni di tipo. Questa pratica può essere fonte di ambiguità e quindi di errore. Il lettore si attenga a quanto detto a proposito delle conversioni implicite ed esplicite di tipo.

12.4 Campi

I *campi* sono un costrutto del C che consente di far riferimento in modo simbolico ai singoli bit di una variabile:

```
struct {
        unsigned k: 1;
        unsigned j: 6;
    }  var_bit_bit;
```

La variabile `var_bit_bit` occupa uno spazio di memoria pari alla somma dei bit utilizzati da ogni campo arrotondata alla *word* (parola) della macchina. In altri termini, supponendo che la parola della macchina sia di 16 bit, si ha che:

- • il campo `k` di `var_bit_bit` occupa 1 bit;
- • il campo `j` di `var_bit_bit` occupa 6 bit;
- • 9 bit rimangono inutilizzati.

I campi possono essere trattati alla stregua dei membri di una struttura e quindi essere utilizzati in espressioni come la seguente

```
var_bit_bit.k = 1;
```

usata per esempio per alzare a uno il flag `k` della variabile `var_bit_bit`. La sintassi generale dei campi è:

```
struct {
        tipo_campo nome_campo1 : numero_bit_1;
        tipo_campo nome_campo2 : numero_bit_2;
        ...
        tipo_campo nome_campoN : numero_bit_N;
    } var_1, var_2, var_N;
```

oppure equivalentemente:

```
struct nome_struct {
        tipo_campo nome_campo1 : numero_bit_1;
```

```
        tipo_campo nome_campo2 : numero_bit_2;
        ...
        tipo_campo nome_campoN : numero_bit_N;
     };
     struct nome_struct var_1, var_2, var_N;
```

I campi non hanno incontrato un grande successo presso i programmatori C, poiché di fatto non producono ottimizzazione né di tempo né di spazio, cosicché al loro posto si preferiscono le classiche operazioni bit a bit.

12.5 typedef

La parola chiave `typedef` viene usata per assegnare un alias a un qualsiasi tipo, fondamentale e derivato. Con `typedef` non si definisce un nuovo tipo all'interno del programma, ma più semplicemente si introduce un nome che corrisponde a uno dei tipi definiti. Per esempio, con le istruzioni

```
typedef char * Stringa;
     Stringa s1, s2;
```

dapprima il tipo `char *`, cioè il tipo puntatore a carattere, viene ribattezzato `Stringa`. Successivamente si definiscono due variabili, `s1` e `s2`. La sintassi di `typedef` è

```
typedef nome_tipo nuovo_nome_tipo;
```

dove *`nome_tipo`* è il nome simbolico del tipo che si vuol ribattezzare e *`nuovo_nome_tipo`* è il nome che si associa a *`nome_tipo`*. Ribattezzare un tipo può essere molto utile soprattutto per rendere più leggibile un programma, e per evitare espressioni altrimenti complesse. Per esempio con il frammento di programma

```
typedef char * Stringa;
     Stringa p;

     int strlen(Stringa);
     p = (Stringa)malloc(100);
```

si introduce il nome `Stringa` per rappresentare un generico puntatore a carattere. Da questo punto in avanti `Stringa` può essere usato come nome di un qualsiasi tipo; infatti ne facciamo uso nella definizione della variabile `p`, del parametro formale della funzione `strln` e nella conversione esplicita di tipo che alloca 100 caratteri e li fa puntare da `p`.

L'uso di `typedef` può risultare utile anche quando si ha a che fare con le strutture. Per esempio si potrebbe avere

```
struct automobile {
        char *marca;
        char *modello;
        int venduto;
     };
     typedef struct automobile Auto;
     Auto car1, car2, car3;
```

In tal modo si evita di ripetere la parola chiave `struct` nella definizione delle variabili `car1`, `car2` e `car3` e al contempo si rende molto espressivo il codice sorgente.
Si osservi come si sia implicitamente mantenuta la convenzione di usare la prima lettera maiuscola nella ridefinizione del nuovo nome di un tipo. In tal modo è possibile riconoscere immediatamente i tipi introdotti dal programmatore.

12.6 Tipi derivati composti

Uno dei punti di forza del C è la capacità di combinare insieme tipi derivati e fondamentali per ottenere strutture dati complesse a piacere, in grado di modellare entità del mondo reale. Per esempio

```
Auto salone[100];
```

definisce un array di 100 variabili di tipo `Auto`, che potrebbe essere usato per mantenere memoria di tutti i tipi di auto e del relativo venduto, ponendo un limite massimo di 100 autovetture disponibili.
Il C è dunque flessibile e permette la costruzione di tipi derivati complessi. Questa caratteristica, comoda nella fase di creazione di un programma, può tuttavia essere fonte di problemi soprattutto quando si rilegge un programma scritto da altri. Per esempio, una dichiarazione del tipo

```
char *(*(*var)())[10];
```

potrebbe risultare non immediatamente comprensibile. Esistono per fortuna delle semplici regole che consentono di interpretate una volta per tutte qualsiasi tipo di dichiarazione. Infatti una dichiarazione di una variabile va sempre letta dall'interno verso l'esterno secondo i seguenti passi:

1. 1. si individua all'interno della dichiarazione l'identificatore della variabile e si controlla se alla sua destra ci sono parentesi aperte, tonde o quadrate;
2. 2. si interpretano le eventuali parentesi tonde o quadrate, e poi si guarda alla sinistra per vedere se c'è un asterisco;
3. 3. se si incontra durante un qualsiasi passo una parentesi tonda chiusa, si torna indietro e si riapplicano le regole 1 e 2 a tutto ciò che si trova all'interno delle parentesi;
4. 4. infine si applica lo specificatore di tipo.

Proviamo subito ad applicare queste regole al precedente esempio:

```
char *(*(*frog)()) [10];
      7   6 4 2 1   3     5
```

I vari passi sono etichettati da 1 a 7 e sono interpretati nel modo seguente:

1 l'identificatore di variabile `frog` è un
2 puntatore a
3 una funzione che ritorna un
4 puntatore a
5 un array di 10 elementi, che sono
6 puntatori a
7 valori di tipo `char`

È lasciato al lettore il compito di decifrare le seguenti dichiarazioni:

```
int *frog[5];
     int (*frog)[5];
```

Di tutte le possibili combinazioni di tipi derivati ne esistono alcune più frequentemente usate:

1. 1. tipi derivati composti tramite struttura;
2. 2. tipi derivati composti tramite funzione;
3. 3. tipi derivati composti tramite puntatore.

Per ogni classe di tipo composto studieremo alcune possibili composizioni, facendo particolare riferimento a quelle in cui intervengono puntatori e array. Molte di queste composizioni sono già state trattate nei capitoli e paragrafi precedenti. Altre, come per esempio i puntatori a funzione, sono invece introdotte per la prima volta

12.7 Tipi derivati composti tramite struttura

La struttura aggrega un insieme di membri che possono appartenere a qualsiasi tipo. Per tale motivo la struttura è un costrutto potente che consente di rappresentare come un tutt'uno oggetti di tipo e dimensione diversi. Per esempio, volendo rappresentare il concetto di anagrafe potremmo definire un insieme di strutture nel seguente modo:

```
struct data {
        int giorno;
        char mese [20];
```

```
      int anno;
   };
   struct ind {
      char via[35];
      int numero;
      char interno
      char citta[30];
      char prov[2];
   };
   struct persona {
      char nome[30];
      char cognome[30];
      struct data data_nasc;
      char comune_nasc[30];
      struct ind indirizzo;
      char telefono[10];
      char parentela[2];
   };
```

Le strutture `data`, `ind` e `persona` hanno membri di tipo e dimensione differenti. In particolare si osserva che la struttura `persona` ha dei membri che sono a loro volta delle variabili struttura: `data_nasc` è di tipo `data` e `indirizzo` è di tipo `ind`. In C è possibile definire strutture di strutture purché le strutture che intervengono come membri siano state definite prima della struttura che le contiene.

Questo, ricordando le regole di visibilità di una variabile, è assolutamente normale. Ma cosa succede se una generica struttura `S` ha un membro che a sua volta è una struttura dello stesso tipo `S`? Se per esempio volessimo mantenere una lista di persone ordinata per ordine alfabetico, dovremmo prevedere nella struttura `persona` un membro di tipo persona che dice qual è la prossima persona nell'elenco ordinato. In C, pedrò, una struttura `S` non può contenere un membro di tipo puntatore a `S`. Così il problema della lista ordinata potrebbe essere risolto con:

```
struct persona {
      char nome[30];
      char cognome[30];
      struct data data_nasc;
      char comune_nasc[30];
      struct ind indirizzo;
      char telefono[10];
      char parentela[2];    /* CF capofamiglia, CG coniuge ecc. */
      struct persona *link; /* punta alla persona seguente */
     }
```

Il caso delle strutture ricorsive è molto più comune di quanto si possa pensare. Alcune delle più importanti strutture dati quali le liste, le pile e gli alberi sono rappresentabili mediante strutture ricorsive; a tale proposito il lettore troverà numerosi esempi in successivi capitoli di questo testo.

Se la struttura è lo strumento con cui si rappresenta una classe di oggetti (le automobili, gli impiegati, i nodi di un albero ecc.), aggregando un insieme di membri è logico aspettarsi una molteplicità di elementi, cioè di variabili. Tornando per esempio al caso dell'anagrafe, avrebbe poco senso concepire tale struttura per modellare una sola persona. Piuttosto è significativo rappresentare un insieme di persone, tutte di tipo `anagrafe`, e quindi adoperare il costrutto array:

```
struct persona anagrafe[300];
```

L'identificatore `anagrafe` è un array di 300 variabili di tipo `persona`, ovvero modella un'anagrafe con al più 300 elementi.

Le strutture e i tipi composti con le strutture sono molto importanti nella programmazione C, perché consentono di organizzare in modo razionale i dati del problema in esame. Un programma, però, non descrive solo dati ma anche funzioni. Per esempio, nel caso dell'anagrafe ci preoccuperemo di scrivere funzioni per l'inserimento di una persona, per la sua cancellazione, ricerca e visualizzazione; quindi è logico aspettarsi in un programma C funzioni che lavorano su variabili di tipo struttura. Per passare una variabile di tipo struttura a una funzione occorre far riferimento a un puntatore alla struttura.

Consideriamo a titolo di esempio il programma del Listato 12.2. La nostra anagrafe è rappresentata da un array di 30 elementi di tipo struttura, `anag`. La struttura dati che modella una persona è molto semplice e comprende:

- `cognome`
- `nome`
- `indirizzo`
- `eta`

Listato 12.2 Gestione anagrafica

Il programma, per mezzo della funzione `men_per`, presenta il seguente menu:

```
  ANAGRAFE

1. Immissione Persona
2. Cancellazione Persona
3. Ricerca Persona
4. Visualizza Anagrafe
0. Fine

Scegliere un'opzione:
```

Con la scelta 1, `Immissione Persona`, si lancia la funzione `ins_per` che inserisce i dati di una persona nell'array `anag`; la funzione è molto semplice: non effettua controllo sull'esistenza di una persona nell'array `anag`, ma inserisce dati finché lo consentono le dimensioni dell'array. Con la scelta 2, `Cancellazione Persona`, viene invocata la funzione `can_per` che richiede dapprima all'utente i seguenti dati:

```
CANCELLA PERSONA
     ---------------
     Cognome  :
     Nome  :
     Età  :
```

e poi invoca la funzione `cer_per`. Quest'ultima prende in ingresso le variabili `cognome`, `nome` ed `età` inserite dall'utente, effettua una ricerca sequenziale nell'array `anag`, e se trova la persona corrispondente ai dati forniti in ingresso restituisce il puntatore, `ps`, alla prima occorrenza dell'array che corrisponde ai dati richiesti dall'utente. Il lettore osservi come `cer_per` sia un classico esempio di funzione che restituisce un puntatore a variabile di tipo struttura:

```
struct per *cer_per(char *cg. char *nm, int et)
     {
     ...
     }
```

Se per `cer_per` non trova nessuna persona con i dati richiesti allora restituisce un puntatore `NULL`. Nel caso in cui invece `cer_per` trovi una occorrenza valida, la funzione `can_per` prende il puntatore all'occorrenza e lo passa a `vis_per`, che ha lo scopo di visualizzare i dati della persona che si vuol eliminare. La funzione `vis_per` è un classico esempio di funzione che prende in ingresso un puntatore a struttura:

```
void vis_per(struct per *p)
     {
     ...
     }
```

Solo dopo conferma da parte dell'utente la persona è eliminata da `eli_per`, altra funzione che prende come ingresso un puntatore a variabile struttura. Comportamento analogo a quello di `can_per` presenta la funzione `ric_per`, invocata scegliendo dal menu la voce 3, `Ricerca Persona`. L'ultima opzione del menu permette di visualizzare uno a uno i dati di tutte le persone presenti in `anag`.

12.8 Tipi derivati composti tramite funzione

Delle funzioni abbiamo parlato nei Capitoli 7 e 9. Abbiamo trattato il caso di funzioni che accettano i puntatori (a qualsiasi cosa essi puntino) e il caso di funzioni che ritornano un puntatore, come `cer_per` dell'esempio precedente. In sintesi ricordiamo che:

- • gli array e le strutture debbono essere passati a una funzione tramite puntatore;
- • passando un puntatore a una funzione si effettua un passaggio di parametri per indirizzo e così si risolve il caso di funzioni che devono restituire più di un valore.

Una funzione, poi, ritorna un puntatore tipicamente in seguito a operazioni di ricerca, come speriamo di aver dimostrato con gli esempi. Esiste però un ultimo caso molto importante, che finora non è mai stato trattato: il *puntatore a funzione*.

In C è possibile definire anche un tipo derivato composto "puntatore a funzione":

```
int (*pf)(float);
```

Applicando rigorosamente le regole di interpretazione di una dichiarazione si ha che l'identificatore `pf` è un puntatore a una funzione che ritorna un `int` e ha un solo parametro di tipo `float`. In generale la sintassi di dichiarazione di un puntatore a funzione è

```
tipo_funzione (*nome_puntatore_funzione)(tipo_parametri);
```

dove *`tipo_funzione`* è il tipo della funzione che agisce sui parametri il cui tipo è *`tipo_parametri`* ed è puntata da *`nome_puntatore_funzione`*.

A che cosa serve un puntatore di funzione? Prima di tentare una risposta si consideri che accedere al contenuto di un puntatore funzione tramite il consueto operatore `*` equivale a invocare la funzione:

```
double risultato;
     float base = 3.0;
     double (*pf)(float);
     double cubo(float);

     pf = cubo; /* inizializzazione del puntatore */
     risultato = (*pf)(base);
     printf("Il cubo di %f è %f", base, risultato)
```

L'istruzione

```
risultato = (*pf)(base);
```

è perfettamente equivalente a

```
risultato = cubo(base);
```

Vediamo ora con il Listato 12.3 un esempio completo volto a chiarire l'importanza dei puntatori di funzione.

```
#include <stdio.h>

void dummy(void), fun1(void), fun2(void), fun3(void);

struct voce_menu {
  char *msg;            /* prompt di voce di menu */
  void (*fun)(void);    /* funzione da innescare */
};

/* inizializza in vettore di strutture menu
assegnando il messaggio della voce di menu e
la relativa funzione */

struct voce_menu menu[] = {
  "1. Funzione fun1\n", fun1,
  "2. Funzione fun2\n", fun2,
  "3. Funzione fun3\n", fun3,
  "0. Fine\n", NULL, NULL , NULL
```

```
};

void main()
{
int scelta;
struct voce_menu *p;
int loop = 0;

while(loop==0) {
  for(p=menu; p->msg!=NULL; p++) /* presentazione del menu */
    printf("%s", p->msg);

  printf("\n\nScegliere l'opzione desiderata: ");
  scanf("%d", &scelta);

  if(scelta==0)      /*uscita programma */
    loop = 1;
  else
    /* esecuzione della funzione associata alla scelta */
    (*menu[scelta-1].fun)();
}
}

void fun1(void)
{printf("\n\n Sto eseguendo fun1\n\n\n");}

void fun2(void)
{printf("\n\n Sto eseguendo fun2\n\n\n");}

void fun3(void)
{printf("\n\n Sto eseguendo fun3\n\n\n");}
```

Listato 12.3 Puntatori di funzione per la creazione di un menu

Questo semplice programma presenta il seguente menu:

```
1. Funzione fun1
     2. Funzione fun2
     3. Funzione fun3
     0. Fine

     Scegliere l'opzione desiderata:
```

Scegliendo una qualsiasi delle funzioni da 1 a 3 per mezzo dell'istruzione

```
(*menu[scelta -1].fun)();
```

si lancia il corrispondente programma. Per esempio, con

```
scelta = 1;
```

si raggiunge l'elemento

```
menu[0].fun
```

il cui contenuto è `fun1`. In conseguenza sul video si ottiene il messaggio:

```
Sto eseguendo fun1
```

Il programma del Listato 12.3, oltre a esemplificare l'uso dei puntatori a funzione, è anche un esempio di come attraverso le strutture sia possibile rappresentare qualsiasi oggetto, compreso un menu.

12.9 Tipi derivati composti tramite puntatore

Il puntatore può essere abbinato a qualsiasi tipo, compreso se stesso. In effetti il puntatore a puntatore – o puntatore di puntatore – è un tipo derivato composto tra i più usati in C. Per esempio:

```
char **pp;
```

è la dichiarazione di un puntatore di puntatore a carattere. Un puntatore di puntatore è una variabile abilitata a mantenere l'indirizzo di una variabile puntatore. Per esempio, nel programma

```
# include <stdio.h>
     void main(void)
     {
        int **ppi;
        int a = 3;
        int *pi;
        char *pausa;

        pi = &a;
        ppi = &pi;
        printf("%d", **ppi);
        gets(pausa);
     }
```

la variabile `pi` contiene l'indirizzo della variabile intera `a`, e `ppi` contiene l'indirizzo di `pi`. Conseguentemente `*ppi` corrisponde al contenuto di `pi`, cioè all'indirizzo di `a`, e `**pp1` corrisponde al contenuto di `a`. Infatti l'istruzione

```
printf("%d", **ppi);
```

visualizza il numero 3 (Figura 12.2).

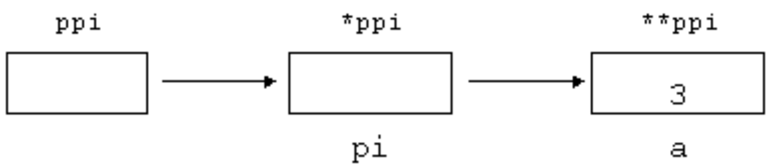


Figura 12.2 Puntatore di puntatore a intero

È naturalmente possibile dichiarare puntatori di puntatori di puntatori. Il programmatore deve però prestare molta attenzione perché una doppia, o peggio ancora, tripla indirezione è difficile da controllare in C; pertanto i puntatori debbono essere usati con parsimonia. Esistono comunque casi in cui è necessario usare una doppia indirezione, cioè un puntatore di puntatore: un tipo derivato composto comunemente usato in C è l'array di puntatori, e per scandire un array di puntatori si fa generalmente uso di un puntatore di puntatore (Listato 12.4).

```
#include <stdio.h>

char *menu[] = {
  "1. Voce di menu 1\n",
  "2. Voce di menu 2\n",
  "3. Voce di menu 3\n",
  ...
  "N. Voce di menu N\n",
  NULL
};

char **ppc = menu;
```

```
main()
{
char *pausa;
while(*ppc!=NULL)
  printf("%s", *ppc++);
gets(pausa);
}
```

Listato 12.4 Un array di puntatori scandito da un puntatore di puntatore

Eseguendo tale programma si ottiene:

```
1.  Voce di Menu 1
     2.  Voce di Menu 2
     3.  Voce di Menu 3
         ...
     N.  Voce di Menu N
```

Come si può osservare, `menu` è un array di puntatori a carattere. Infatti il lettore ricordi che ogni costante stringa riportata tra doppi apici corrisponde all'indirizzo del primo carattere della stringa. Per contrassegnare la fine dell'array di puntatori a carattere `menu` si è usato il puntatore `NULL`. Per visualizzare le varie voci di menu basta passare a `printf` l'indirizzo delle voci di menu raggiungibile tramite `*ppc`. Inoltre `ppc` deve essere incrementato per scandire gli elementi dell'array (Figura 12.3).

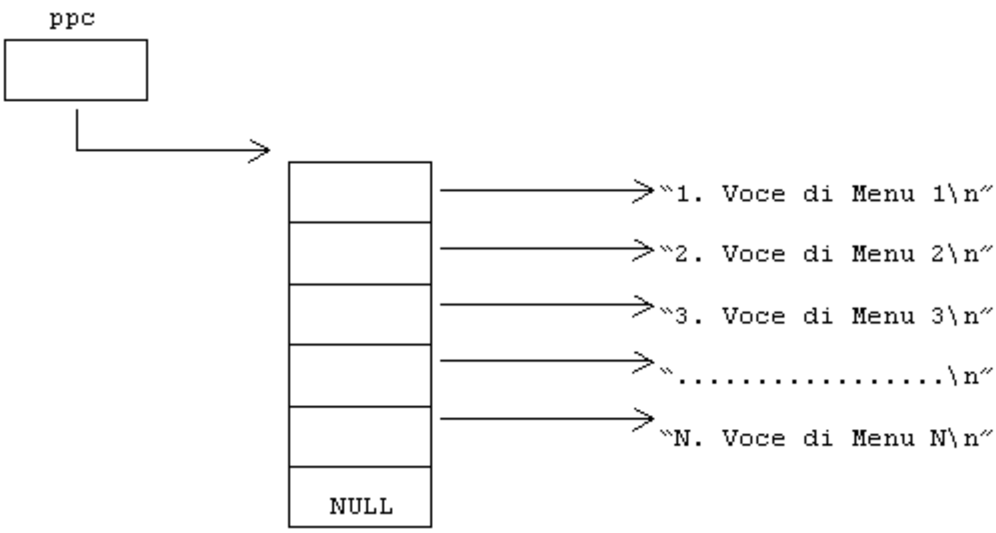


Figura 12.3 Rappresentazione grafica di un array di puntatori

La variabile `menu` è un array di puntatori. La sintassi degli array di puntatori è

*tipo_puntato *nome_array[dim_array];*

Questo tipo derivato composto è particolarmente utile quando si voglia collezionare un insieme di oggetti – in generale – a dimensione variabile, come le stringhe di caratteri. Accanto a un array di puntatori si può trovare un puntatore di puntatore, usato per scandire l'array di puntatori al posto dell'indice dell'array.
Un array di puntatori viene usato anche per passare un numero variabile di parametri alla funzione `main`. Infatti, anche il `main` è una funzione, come abbiamo più volte osservato, e il C mette a disposizione del programmatore un meccanismo attraverso cui passare parametri. Osserviamo il semplice programma:

```
/* prova.c : esempio di programma con parametri sulla linea di
comando                                                          */

     main(int argc, char *argv[])
     {
```

```
      int i;
      for(i=1; i<argc; i++)
         printf(argv[i]);
   }
```

I due parametri formali `argc` e `argv` sono parametri standard e sono gli unici ammessi per la funzione `main`. Essi hanno un preciso significato: `argc` è un `int` che contiene il numero di parametri presenti sulla linea di comando, incluso il nome del programma. Per esempio, lanciando il programma `prova` seguito da un insieme di parametri:

```
C:\>prova uno due tre
```

la variabile standard `argc` vale 4 (infatti sono quattro le stringhe presenti sulla linea di comando); `argv` è un array di puntatori a carattere che contiene gli indirizzi delle quattro stringhe `prova`, `uno`, `due`, `tre` (Figura 12.4).

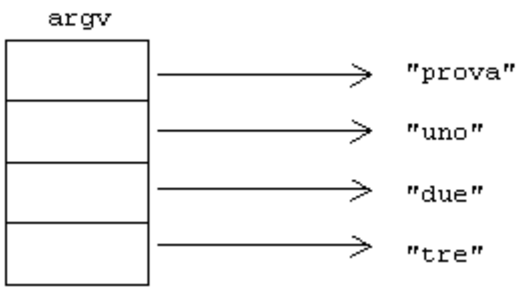


Figura 12.4 Rappresentazione del passaggio di parametri a `main`

Nel programma **`prova.c`** attraverso l'indice `i` si scandisce l'array standard **`argc`** e l'indirizzo delle stringhe `uno`, `due` e `tre` viene iterativamente passato alla funzione `printf`. Dunque lanciando `prova`:

```
C:\>prova uno due tre
```

si ottiene

```
unoduetre
```

12.10 Classificazione delle variabili

Oltre a essere classificata in base al tipo (fondamentale, derivato e derivato composto), in C una variabile può essere anche classificata secondo la visibilità all'interno del programma e secondo la classe di memoria.
Del concetto di visibilità di una variabile abbiamo già trattato a proposito delle funzioni, dove abbiamo imparato a distinguere tra variabili locali e globali. L'ambito di definizione o *scope* di un nome globale si estende dal punto di definizione sino alla fine del file, mentre l'ambito di validità di un nome locale si estende dal punto di definizione sino alla fine del blocco in cui è contenuto.

```
#include <stdio.h>
#include <string.h>

#define DIM 31
#define TAPPO "THE END"

/* semplice struttura che modella una persona */
struct per {
  char cognome[DIM];
  char nome[DIM];
  char ind[DIM];
  int eta;
};
```

```
/* vettore di persone */
struct per anag[] = {
  {"Edison", "Thomas", "Vicolo della Lampadina, 8", 30},
  {"Alighieri", "Dante", "Via del Purgatorio, 13", 21},
  {"More", "Thomas", "Viale Utopia, 48", 39},
  {TAPPO, TAPPO, TAPPO, 0}
};

void vis_per(void);

main()
{
  vis_per();
}

void vis_per(void)
{
char pausa; int i;

for(i=0; strcmp(anag[i].cognome, TAPPO)!=0; i++) {
  printf("\n\n--------------------\n");
  printf("\n\t\tCognome : %s", anag[i].cognome);
  printf("\n\t\tNome : %s", anag[i].nome);
  printf("\n\t\tIndirizzo : %s", anag[i].ind);
  printf("\n\t\tEtà : %d", anag[i].eta);
  printf("\n\n--------------------\n");

  scanf("%c", &pausa);
}
}
```

Listato 12.5 Variabili locali e globali

Consideriamo il Listato 12.5. In questo programma la variabile `anag` è globale mentre le variabili `i` e `pausa` sono locali rispettivamente alla funzione `vis_per`. Finora abbiamo sempre considerato programmi C contenuti in un solo file sorgente. In generale, però, un programma C di media-alta complessità si estende su più file. Per esempio, il programma precedente potrebbe essere diviso su due file, uno contenente il `main` e l'altro la funzione `vis_per` e tutte le altre eventuali funzioni che vorremo far operare su `anag` (Listati 12.6 e 12.7).

```
/* programma principale: MAIN.C */

extern void vis_per(void);

main()
{
  vis_per();
}
```

Listato 12.6 File `MAIN.C`

```
/* file delle funzioni: VIS_PER.C */

#include <stdio.h>
#include <string.h>

#define DIM 31
#define TAPPO "THE END"

/* semplice struttura che modella una persona */
```

```
struct per {
  char cognome[DIM];
  char nome[DIM];
  char ind[DIM];
  int eta;
};

/* vettore di persone */
struct per anag[] = {
  {"Edison", "Thomas", "Vicolo della Lampadina, 8", 30},
  {"Alighieri", "Dante", "Via del Purgatorio, 13", 21},
  {"More", "Thomas", "Viale Utopia, 48", 39},
  {TAPPO, TAPPO, TAPPO, 0}
};

void vis_per(void)
{
char pausa; int i;

for(i=0; strcmp(anag[i].cognome, TAPPO)!=0; i++) {
  printf("\n\n--------------------\n");
  printf("\n\t\tCognome : %s", anag[i].cognome);
  printf("\n\t\tNome : %s", anag[i].nome);
  printf("\n\t\tIndirizzo : %s", anag[i].ind);
  printf("\n\t\tEtà : %d", anag[i].eta);
  printf("\n\n--------------------\n");

  scanf("%c", &pausa);
}
}
```

Listato 12.7 File `VIS_PER.C`

Il nome simbolico `vis_per` dichiarato nel file `MAIN.C` con

```
extern void vis_per(void);
```

non ha all'interno del file una corrispondente definizione. Per tale motivo il nome è dichiarato come `extern`. La parola chiave `extern` si usa per informare il compilatore del fatto che la definizione del simbolo dichiarato `extern` è presente in qualche altro file. Nel nostro esempio la definizione della funzione `vis_per` si trova nel file `VIS_PER.C`.

Possiamo ora definire con esattezza il concetto di variabile globale.

In C una variabile globale è visibile in tutti i file sorgenti che compongono il programma. Per poter far riferimento a una variabile globale che si trova in un altro file occorre però usare la dichiarazione `extern`. Sfortunatamente spesso il *linker* non controlla la congruità fra il tipo e la dimensione della variabile `extern` e quello della corrispondente variabile globale: tale congruenza deve essere mantenuta dal programmatore. Se per esempio avessimo scritto per errore in `MAIN.C`

```
extern char vis_per(int); /* invece di void vis_per(void) */
```

il compilatore e il *linker* non avrebbero avvertito e in esecuzione il programma avrebbe avuto dei problemi.

Se vogliamo fare in modo che una variabile globale – cioè definita fuori di un blocco – rimanga globale ma solo all'interno del file in cui è definita senza diventare nota a tutti gli altri file, allora la variabile globale deve essere dichiarata `static`. Per esempio, se volessimo rendere visibile solo al file `VIS_PER.C` la variabile globale `anag` dovremmo scrivere come nel Listato 12.8.

```
#include <stdio.h>
#include <string.h>

#define DIM 31
```

```
#define TAPPO "X_Y_Z"

struct per {
  char cognome[DIM];
  char nome[DIM];
  char ind[DIM];
  int eta;
};

/* vettore di persone */
static struct per anag[] = {
  {"Edison", "Thomas", "Vicolo della Lampadina, 8", 30},
  {"Alighieri", "Dante", "Via del Purgatorio, 13", 21},
  {"More", "Thomas", "Viale Utopia, 48", 39},
  {TAPPO, TAPPO, TAPPO, 0}
};

void vis_per(void)
{
char pausa; int i;

for(i=0; strcmp(anag[i].cognome, TAPPO)!=0; i++) {
  printf("\n\n--------------------\n");
  printf("\n\t\tCognome : %s", anag[i].cognome);
  printf("\n\t\tNome : %s", anag[i].nome);
  printf("\n\t\tIndirizzo : %s", anag[i].ind);
  printf("\n\t\tEtà : %d", anag[i].eta);
  printf("\n\n--------------------\n");

  scanf("%c", &pausa);
}
}
```

Listato 12.8 Esempio di variabili `static`

```
static struct per anag[];
```

Allora se nel file `MAIN.C` tentassimo di far riferimento alla variabile `anag` mediante una dichiarazione del tipo:

```
extern struct per anag[];
```

non otterremmo altro effetto se non quello di provocare un errore in fase di link.

Come abbiamo anticipato, oltre a essere classificate in base al tipo le variabili C sono classificate anche in base alla *classe di memoria*. La classe di memoria è un attributo dell'oggetto, cioè dell'area di memoria, associato a ogni variabile il quale stabilisce quando una variabile nasce e quando muore, cioè quando è deallocato il corrispondente oggetto. A questo proposito in C esistono due classi di memoria che si riferiscono a due variabili:

- • variabili automatiche;
- • variabili non automatiche.

Le variabili automatiche sono tutte le variabile locali, cioè dichiarate all'interno di un blocco, che non possiedono l'attributo `static`. Tipicamente variabili automatiche sono gli argomenti di una funzione e le variabili locali di una funzione. Gli oggetti relativi alle variabili automatiche vengono allocati all'atto dell'invocazione della funzione e deallocati quando la funzione termina. Così, nell'esempio:

```
int f(int x, char b)
     {
       int a, b, c;
       char *pc;
       ...
     }
```

le variabili `x`, `y`, `a`, `b`, `c` e `pc` sono tutte automatiche, ovvero vengono create all'atto della chiamata di `f` e sono distrutte quando `f` termina e ritorna il controllo al programma chiamante. Le variabili automatiche, essendo in un certo senso "provvisorie", sono allocate nello *stack*. Poiché le dimensioni dello stack sono limitate, si sconsiglia di utilizzare funzioni con molte variabili automatiche. Per esempio è conveniente evitare funzioni che definiscono al proprio interno strutture dati che allocano grosse fette di memoria. Se invece si vuole estendere il ciclo di vita di una variabile locale, magari definita all'interno di una funzione, anche a dopo che la funzione è ritornata, basta dichiararla `static`. Nel frammento di codice seguente:

```
int f(int x, char b)
     {
        static int a;
        int b, c;
        char *pc;
        ...
     }
```

la variabile `a` è "statica" o – come si dice – non automatica. Essa nasce all'atto dell'esecuzione dell'intero programma e resta attiva per tutto il tempo in cui il programma è in esecuzione.

Le variabili non automatiche sono molto usate per tutte le funzioni che hanno necessità di mantenere memoria di un valore che esse stesse hanno definito. Si consideri per esempio il caso di un semplice generatore di numeri casuali (Listato 12.9). L'esecuzione del programma produce il seguente risultato:

```
Il numero casuale 1 è 18625
     Il numero casuale 2 è 16430
     Il numero casuale 3 è 8543
     Il numero casuale 4 è 43172
     Il numero casuale 5 è 64653
     Il numero casuale 6 è 2794
     Il numero casuale 7 è 27083
     Il numero casuale 8 è 3200
     Il numero casuale 9 è 23705
     Il numero casuale 10 è 34534
```

```
#include <stdio.h>

#define FATTORE 25173
#define MODULO         65535
#define INCREMENTO     13849
#define SEME_INIZIALE      8
#define LOOP 10

unsigned rand(void);

void main()
{
int i;
for(i=0; i<LOOP; i++)
   printf("Il numero casuale %d è %6u\n\n", i+1, rand());
}

unsigned rand(void)
{
static unsigned seme = SEME_INIZIALE;
seme = (FATTORE*seme+INCREMENTO) % MODULO;
return(seme);
}
```

Listato 12.9 Un generatore di numeri casuali

La funzione `rand` è eseguita per 10 volte. La prima volta la variabile locale `seme` è inizializzata con il valore `SEME_INIZIALE`. Poi la funzione calcola il nuovo valore di `seme` pari a 18625 e lo restituisce alla funzione `printf`

che lo visualizza. Quando rand è eseguita per la seconda volta, essendo rimasta in vita la variabile seme anche dopo il ritorno della funzione, il valore di seme è 18625 e il successivo valore è calcolato in 16430. Il procedimento prosegue per tutte le altre iterazioni. Se la variabile seme non fosse stata dichiarata static la funzione rand avrebbe ricalcolato per 10 volte il medesimo valore 18625 (provare per credere).
In generale, si considerano variabili non automatiche le variabili che hanno un ciclo di vita che si estende per tutta la durata dell'esecuzione del programma. Secondo tale definizione, allora, variabili non automatiche sono non soltanto le variabili locali dichiarate static ma anche tutte le variabili globali.
Il lettore presti attenzione a come viene usata la parola chiave static in C. Infatti abbiamo visto che ne esistono due possibili usi:

1. 1. relativamente alle variabili globali l'attributo static fa sì che la variabile sia locale, ma limitatamente al file in cui è definita senza essere visibile a tutti gli altri file del programma;
2. 2. relativamente alle variabili locali l'attributo static fa sì che il valore della variabile venga mantenuto anche dopo l'uscita del blocco in cui la variabile è definita, e si conservi sino alla fine del programma.

Per concludere con la classificazione delle variabili, osserviamo che esiste un altro attributo che può caratterizzare una variabile: l'attributo register. Questo è sempre riferito a variabili automatiche e dice al compilatore di allocare il relativo oggetto direttamente nei registri macchina dell'unità di elaborazione centrale (CPU). Per mezzo della direttiva register, in teoria, si velocizzano le operazioni sulle relative variabili. Per esempio la funzione:

```
void strcpy(register char *S, register char *t)
      {
        while(*s++ = *t++);
      }
```

viene in teoria eseguita più velocemente di

```
void strcpy(char *s, char *t)
      {
        while(*s++ = *t++);
      }
```

In realtà i moderni compilatori sono talmente ottimizzati che da soli provvedono ad allocare nei registri di memoria le variabili più frequentemente usate. Addirittura talvolta, esagerando con il numero delle variabili dichiarate register, si può sortire l'effetto contrario e rendere più inefficiente il programma. Infatti il numero dei registri macchina è limitato, e spingere il compilatore ad allocare molte variabili con pochi registri significa perdere tempo in una gestione di scarsa utilità.

12.11 Esercizi

1. Scrivere un programma per l'inserimento e visualizzazione di un gruppo di automobili descritte da marca, modello e numero di unità vendute. Il gruppo di automobili è inserito dall'utente del programma.

2. Modificare il programma precedente considerando che il modello di un'automobile è descritto da un codice, una descrizione e un anno.

3. Scomporre in funzioni il programma di inserimento e visualizzazione di un gruppo di automobili.

4. Riscrivere il programma dell'Esercizio 1 come nell'esercizio precedente, ma mettendo la funzione `main` in un file sorgente e le funzioni di inserimento e visualizzazione in un altro.

5. Interpretare le seguenti dichiarazioni:

```
1      long *pony(long, long);
2      long (*dog)(long, long);
3      struct cat {
          int a;
          char b;
       } (*dog[5]) (struct cat, struct cat);
4      double (*mpuse(double(*)[3]))[3];
5      union rabbit {
          int x;
          unsigned y;
          } **frog[5][5];
6      union rabbit *(*frog[5])[5];
```

13.1 Apertura e chiusura di file

Il C, al contrario di altri linguaggi, offre soltanto alcune funzioni di base per operare sui file: non esiste il concetto di record e di chiave di ricerca; non esistono pertanto neppure le classiche funzioni di inserimento, modifica, cancellazione e ricerca di record all'interno di un file. È consentito operare solamente sui file costituiti da sequenze lineari di byte, mediante funzioni per la creazione, rimozione di file, lettura e scrittura di byte da/su file.
La prima operazione da eseguire prima di poter scrivere o leggere da file è l'apertura:

```
#include <stdio.h>

main()
{
  FILE      *fp;
  fp = fopen("ordini", "r");
  ...
```

Il programmatore deve includere `stdio.h`, dove è contenuta la definizione del tipo derivato `FILE`, prima di poter utilizzare le funzioni di accesso ai file. La funzione `fopen` apre il file di nome `ordini`; l'operazione di apertura serve ad associare la variabile puntatore `fp`, nota con il nome di *file pointer*, al file di nome `ordini`. Dopo che il file è stato aperto, il programmatore dovrà utilizzare il file pointer `fp` per poter scrivere e leggere le informazioni memorizzate al suo interno. Nel caso in cui si verifichi un errore, per esempio perché il file `ordini` non esiste, la funzione `fopen` ritorna un file pointer `NULL`.

Dopo avere terminato le operazioni di lettura e/o scrittura è necessario eseguire l'operazione di chiusura del file:

```
fclose(fp);
```

La chiusura del file garantisce che tutti i dati scritti nel file `ordini` siano salvati su disco; infatti molto spesso il sistema operativo, per ottimizzare le prestazioni del programma, ritarda le scritture sulla memoria di massa mantenendo le informazioni temporaneamente in memoria centrale.
Al momento dell'apertura è sempre necessario specificare, oltre al nome del file, anche il tipo di operazione che si desidera eseguire. Nell'esempio precedente il parametro "`r`" stava a indicare che il file doveva essere aperto in lettura (*read*). In generale, le possibili modalità di apertura di un file, specificate dal secondo parametro di `fopen`, sono le seguenti.

`"r"`	*sola lettura* – sul file sarà possibile eseguire soltanto operazioni di lettura e quindi non sarà possibile effettuarvi delle scritture. Se il file non esiste la funzione `fopen` ritornerà il codice di errore `NULL`.
`"w"`	*sola scrittura* – sul file sarà possibile eseguire solamente operazioni di scrittura, quindi non operazioni di lettura. Se il file al momento dell'apertura non esiste sarà automaticamente creato, in caso contrario il contenuto del file preesistente sarà perso.
`"r+"`	*lettura e scrittura* – sul file sarà possibile eseguire operazioni sia di lettura sia di scrittura. Se il file non esiste la funzione `fopen` ritornerà il codice di errore `NULL`.
`"w+"`	*scrittura e lettura* – sul file sarà possibile eseguire operazioni sia di scrittura sia di lettura . Se il file non esiste verrà automaticamente creato, in caso contrario il contenuto del file preesistente verrà perso.
`"a"`	*append* – sul file sarà possibile eseguire soltanto operazioni di scrittura a fine file: tutte le scritture verranno sempre eseguite in coda al contenuto attuale del file. Se il file non esiste verrà creato automaticamente, in caso contrario il contenuto del file preesistente verrà mantenuto.
`"a+"`	*lettura e append* – sul file sarà possibile eseguire operazioni sia di lettura sia di scrittura. Se il file non esiste verrà automaticamente creato, in caso contrario il contenuto del file preesistente verrà mantenuto.

13.2 Lettura e scrittura su file

Dopo aver imparato come si apre e si chiude un file, vediamo come utilizzare le funzioni che consentono di leggere il contenuto di un file e di modificarlo. Partiamo al solito con un esempio:

```
int elementi, dimensione;
char buf[100];
FILE *fp;
int n;
...
elementi = 100;
dimensione = 1;

n = fread(buf, dimensione, elementi, fp);
```

La funzione `fread` legge 100 caratteri dal file `fp` e li trasferisce nel vettore `buf`. I parametri della funzione `fread` sono:

`buf`	è il vettore dove devono essere trasferite le informazioni lette dal file
`dimensione`	rappresenta la dimensione in byte di un elemento del vettore
`elementi`	indica il numero di elementi del vettore
`fp`	è il file da leggere

Il valore di ritorno della `fread` indica al programmatore il numero di elementi letti dal file; tale numero può non coincidere con il numero di elementi del vettore `buf`, come nel caso in cui il file sia vuoto e contenga un numero di elementi inferiore a quello di `buf`. Se il valore di ritorno assume valore negativo significa che è stato commesso qualche errore, per esempio il file non è stato aperto. Poiché in generale un file può avere notevoli dimensioni, sarebbe impensabile poterlo leggere con una sola chiamata a `fread`: quest'ultima può essere ripetuta più di una volta, leggendo così a ogni chiamata soltanto una limitata porzione del file.
La operazioni di lettura accedono al file in maniera sequenziale e mantengono traccia del punto in cui si è arrivati nella lettura. Dopo l'apertura in lettura il puntatore si trova posizionato all'inizio del file; a ogni chiamata a `fread` il

puntatore si sposta in avanti di un numero di byte pari a quelli che sono stati letti e trasferiti nella memoria centrale. Quando tutto il contenuto del file è stato letto la funzione `fread` ritorna il valore `0` per indicare che il puntatore è ormai posizionato a fine file; ogni ulteriore tentativo di lettura fallirà e `fread` continuerà a restituire il valore di ritorno `0`.
Scriviamo ora un semplice programma che conta il numero di caratteri contenuti nel file di nome `clienti` (Listato 13.1).

```
/* Determina il numero di caratteri di un file esistente */

#include <stdio.h>

main()
{
char buf[100];   /* Buffer per la lettura */
FILE *fp;        /* File pointer */
long nc;         /* Contatore caratteri */
int n;           /* Numero caratteri letti con fread() */
int fine_file =0;/* Indica la fine della lettura del file */

fp = fopen("clienti", "r");/* Apertura del file clienti */

if( fp == NULL )
/* Si è verificato un errore: il file non esiste */
  printf("Errore : il file ordini non esiste\n");
else {
  nc = 0L;                 /* Inizializza il contatore */
  do {                     /* Ciclo di lettura */
    /* Legge 100 caratteri dal file ordini */
    n = fread(buf, 1, 100, fp);
    if(n==0)               /* Controllo di fine file */
      fine_file = 1;
    nc += n;               /* Incremento del contatore */
   }
   while(fine_file==0);

   fclose(fp);            /* Chiusura del file clienti */
   printf("Il file clienti contiene %ld caratteri\n", nc);
}
}
```

Listato 13.1 Conta il numero di caratteri nel file `clienti`

Probabilmente i programmatori C avrebbero preferito scrivere il ciclo di lettura in questo modo:

```
for(;;) {
  n = fread(buf, 1, 100, fp);
  if(n==0) break;
  nc += n;
}
```

dove non viene utilizzata la variabile `fine_file`, ma l'uscita dal ciclo di lettura è effettuata per mezzo dell'istruzione `break`, che passa il controllo alla prima istruzione successiva al ciclo stesso.
La funzione per scrivere su di un file, `fwrite`, è analoga a `fread`:

```
int elementi, dimensione;
char buf[100];
FILE *fp;
int n;
...
elementi = 100;
```

```
dimensione = 1;

n = fwrite(buf, dimensione, elementi, fp);
```

`fwrite` scrive i 100 caratteri del vettore `buf` nel file `fp`. I parametri della funzione sono i seguenti:

`buf`	è il vettore che contiene i dati che devono essere memorizzati nel file `fp`
`dimensione`	rappresenta la dimensione in byte di un elemento del vettore
`elementi`	indica il numero di elementi del vettore
`fp`	è il file dove devono essere memorizzati i dati

Il valore di ritorno della `fwrite` indica al programmatore il numero di elementi che sono stati memorizzati nel file; tale numero può non coincidere con quello degli elementi del vettore `buf`, per esempio nel caso in cui il file abbia raggiunto la massima dimensione ammessa. Se il valore di ritorno assume valore negativo significa che è stato commesso qualche errore, per esempio il file non è stato aperto.
Scriviamo ora due semplici programmi: il primo acquisisce una stringa da tastiera e la memorizza all'interno del file di nome `fornitori` (Listato 13.2); il secondo copia il contenuto del file `ordini` nel file `ordini.bak` (Listato 13.3)
.

```
/* Scrittura di una stringa in un file */

#include <stdio.h>
#include <string.h>

main()
{
  char buf[100];         /* Buffer */
  FILE *fp;              /* File pointer */
  int len;

  /* Legge da tastiera il nome del fornitore */
  printf("Inserisci un fornitore : ");
  scanf("%s",buf);
  len = strlen(buf);
  fp = fopen("fornitori", "w"); /* Crea il file fornitori */

  /* Memorizza il nome del fornitore nel file */
  fwrite(buf, 1, len, fp);
  fclose(fp);                   /* Chiude il file */
}
```

Listato 13.2 Programma per l'acquisizione di una stringa da tastiera e sua scrittura in un file

```
/* Copia di un file su un altro */

#include <stdio.h>

main()
{
FILE *fpin, *fpout;     /* File pointer */
char buf[512];          /* Buffer dati */
int n;

fpin = fopen("ordini","r");      /* Apre ordini in lettura */
if(fpin!=NULL) {
fpout = fopen("ordini.bak", "w");      /* Crea  ordini.bak */
  if(fpout!=NULL) {    /* ordini.bak creato correttamente? */
    for(;;) {               /* Copia ordini in ordini.bak */
```

```
      n = fread(buf, 1, 512, fpin);       /* Legge ordini */
      if( n == 0 ) break;       /* controllo di fine file */
      fwrite(buf, 1, n, fpout);   /* Scrive in ordini.bak */
    }
    fclose(fpin);      /* Chiude il file ordini */
    fclose(fpout);     /* Chiude il file ordini.bak */
  }
  else {
    printf("Il file ordini.bak non può essere aperto\n");
         fclose(fpin);          /* Chiude il file ordini */
  }
}
else
/* Errore di apertura */
  printf("Il file ordini non esiste\n");
}
```

Listato 13.3 Programma per la copia di un file su un altro

13.3 Posizionamento del puntatore

In C è possibile operare sui file di byte non solo in modo strettamente sequenziale ma anche con modalità *random*. La funzione `fseek` consente infatti di muovere il puntatore di lettura e/o scrittura in una qualunque posizione all'interno del file.
`fseek` si usa come illustrato nel seguente esempio:

```
int err, mode;
FILE *fp;
long n;

mode = 0;
n = 100L;

err = fseek(fp, n, mode);
```

in cui il file pointer `fp` è posizionato sul centesimo byte. I parametri della funzione `fseek` hanno il seguente significato:

`fp`	è il file pointer;
`n`	indica di quanti byte il file pointer deve essere spostato; se `n` è negativo significa che il file pointer deve essere spostato indietro invece che in avanti;
`mode`	indica a partire da quale posizione muovere il file pointer; se `mode` vale `0` significa che lo spostamento è a partire dall'inizio, se vale `1` è dalla posizione corrente e se, infine, vale `2` è a partire dalla fine del file.

Il valore di ritorno della funzione `fseek` è negativo se si è verificato un errore, maggiore o uguale a 0 in caso contrario.
Il C mette a disposizione una funzione per conoscere la posizione corrente del file pointer; tale funzione prende il nome di `ftell` e si usa nel seguente modo:

```
FILE *fp;
long n;
...
n = ftell(fp);
```

La funzione `ftell` ritorna la posizione corrente del file pointer; se si verifica un errore, per esempio se il file non è stato aperto, `ftell` ritorna un valore negativo. Scriviamo un semplice programma che visualizza la dimensione di un file utilizzando la funzione `fseek` e la funzione `ftell` (Listato 13.4).

```
/* Determinazione del numero di caratteri di un file
con fseek() e ftell()                                    */

#include <stdio.h>

main(int argc, char **argv)
{
FILE *fp;
long n;

if( argc < 2 )
  printf("File non specificato\n");
else {
  fp = fopen(argv[1], "r");      /* Apertura del file */

  if( fp != NULL ) {                /* Il file esiste? */
    fseek(fp,0L,2);    /* Puntatore alla fine del file */
n = ftell(fp);    /* Lettura posizione del puntatore */
    fclose(fp);                    /* Chiusura del file */
    printf("La dimensione del file è %ld\n", n);
  }
  else
    printf("Errore : il file %s non esiste\n", argv[1]);
}
}
```

Listato 13.4 Visualizzazione della dimensione di un file con `fseek` e `ftell`

13.4 Lettura e scrittura formattata

Nel Capitolo 11 avevamo approfondito l'uso delle funzioni di libreria `printf` e `scanf`, che consentono di scrivere a video e di acquisire dati da tastiera, e introdotto le funzioni `fprintf` e `fscanf`; riprendiamo queste ultime in relazione alla scrittura e lettura su file:

- • `fprintf` scrive sui file in modo formattato
- • `fscanf` legge da file in modo formattato

La funzione `fprintf` si usa come nell'esempio seguente:

```
FILE *fp;
int i;
...
i = 2;
fprintf(fp, "Il valore di i è %d\n", i);
```

dove `fp` rappresenta il file sul quale si desidera scrivere; un uso analogo è quello che si fa di `fscanf`:

```
FILE *fp;
int i;
...
fscanf(fp,"%d", &i);
```

dove `fp` rappresenta il file dal quale si desidera leggere.
Altre due funzioni assai utili per poter operare sui file sono `fgets` e `fputs` che, rispettivamente, leggono e scrivono una riga su file, dove per riga si intende una sequenza di caratteri terminata dal carattere di newline ("`\n`"). La funzione `fgets` opera secondo questa falsariga:

```
char buf[100];
FILE *fp;
char *s;
int n;
...
n = 100;
s = fgets(buf, n, fp);
```

dove `buf` rappresenta il vettore in cui la riga letta dal file `fp` deve essere trasferita e `n` rappresenta la dimensione del vettore `buf`; la funzione `fgets` ritorna il puntatore a carattere: tale valore coincide con l'indirizzo di `buf` se tutto è andato a buon fine, mentre vale `NULL` nel caso in cui si sia verificato un errore o il file pointer si trovi posizionato a fine file.
La funzione `fgets` non si limita a trasferire una riga nel vettore `buf` ma aggiunge automaticamente a fine riga il carattere di fine stringa "`\0`".
La funzione `fputs` scrive una riga in un file. La modalità di uso è la seguente:

```
char buf[100];
FILE *fp;
...
fputs(buf,fp);
```

dove `buf` è il vettore che contiene la riga da scrivere sul file `fp` terminata con il carattere di fine stringa "`\0`".
Scriviamo ora un semplice programma di esempio che conta il numero di righe di un file (Listato 13.5).

```
/* Determinazione del numero di linee contenute in un file.
   Ogni linea è definita dal carattere di newline \n   */

#include <stdio.h>

main(int argc, char **argv)
{
char buf[100];
int linee;
FILE *fp;

if( argc < 2 )
  printf("Errato numero di parametri\n");
else {
  fp = fopen(argv[1], "r");             /* Apre il file */
  if(fp!= NULL) {                    /* Il file esiste? */
    linee = 0;        /* Inizializza contatore di linea */
    for(;;) {              /* Ciclo di lettura da file */
      if( fgets(buf,100,fp) == NULL )
        break;                              /* Fine file */
      linee++;            /* Incrementa contatore linee */
    }
    fclose(fp);                        /* Chiude il file */
    printf("Il file contiene %d linee\n", linee);
  }
  else
    printf("Il file %s non esiste\n",argv[1]);
}
}
```

Listato 13.5 Programma che conta il numero di righe di un file

Esistono inoltre due funzioni per leggere e scrivere un singolo carattere da file: `fgetc` e `fputc`.
La funzione `fgetc` opera nel seguente modo:

```
FILE *fp;
int c;
...
c = fgetc(fp);
```

dove `fp` rappresenta il file pointer. Il valore di ritorno della funzione `fgetc` è di tipo `int` e non di tipo `char` perché ha un duplice significato: se coincide con la costante simbolica `EOF`, definita nel file di include `stdio.h`, significa che il file pointer è posizionato a fine file e quindi nessun carattere è stato letto, mentre in caso contrario il valore di ritorno rappresenta il carattere letto dal file. La costante simbolica `EOF` è definita in modo tale che non possa mai eguagliare alcun carattere; solitamente vale –1.
La funzione `fputc` lavora nel seguente modo:

```
FILE *fp;
int c;
...
c = 'A;
fputc(c,fp);
```

dove `c` rappresenta il carattere da scrivere sul file pointer `fp`.
Scriviamo ora un programma che conta il numero di caratteri numerici contenuti in un file (Listato 14.6).

```
/* Determinazione del numero di caratteri numerici
   (cifre decimali) presenti in un file          */

#include <stdio.h>

main(int argc, char **argv)
{
FILE *fp;
int c;
int nc;

if( argc < 2 )
  printf("Errato numero di parametri\n");
else {
  fp = fopen(argv[1],"r");                  /* Apre il file */
  if(fp!=NULL) {                         /* Il file esiste? */
    nc = 0;                        /* Inizializza il contatore */
    while((c = fgetc(fp)) != EOF)       /* Ciclo di lettura */
      if(c>='0' && c<='9') nc++; /* Incrementa il contatore */
    fclose(fp);                            /* Chiude il file */
    printf("Il numero di caratteri numerici è: %d\n", nc);
  }
  else
    printf("Il file %s non esiste\n",argv[1]);
}
}
```

Listato 13.6 Programma che conta il numero di caratteri numerici contenuti in un file

Spesso il programmatore desidera sapere semplicemente se il file pointer si trova posizionato a fine file; a tale scopo ha a disposizione la funzione `feof`, la cui modalità d'uso è:

```
int ret;
FILE *fp;
...
ret = feof(fp);
```

dove fp rappresenta il file pointer. Il valore di ritorno ret conterrà il numero 1 se il file pointer è posizionato a fine file e il numero 0 in caso contrario.
Tutte le funzioni sin qui descritte lavorano in modalità bufferizzata: molte operazioni di scrittura non sono fisicamente eseguite su disco ma rimangono temporaneamente memorizzate in memoria centrale per ottimizzare le prestazioni. Soltanto una scrittura su molte viene realmente eseguita. Lo stesso discorso vale per le operazioni di lettura, poiché il C durante le letture fisiche da disco cerca di trasferire in memoria centrale una quantità di informazioni superiore a quella richiesta dal programmatore, con l'obiettivo di evitare altre letture fisiche successive, poiché i dati che verranno richiesti sono già stati pre-letti.
Il C esegue queste operazioni in maniera del tutto trasparente, utilizzando dei buffer di memoria la cui dimensione è ottimizzata sulla base delle caratteristiche del sistema operativo ospite e della memoria di massa. Al momento della chiusura di un file o quando il programma termina la sua esecuzione, il contenuto di tutti i buffer viene scaricato su disco. La costante simbolica BUFSIZ, contenuta nel file di include stdio.h, rappresenta la dimensione in byte dei buffer per le operazioni di lettura e scrittura.
Talvolta il programmatore ha la necessità di scaricare su disco tutte le scritture che sono rimaste temporaneamente memorizzate nei buffer. La funzione fflush assolve il compito di scaricare il contenuto del buffer associato a un file pointer:

```
#include <stdio.h>

main()
{
FILE *fp;

fp = fopen("ordini", "w");
fprintf(fp,"Giovanni Fuciletti");

fflush(fp);
...
```

Nel caso in cui il programmatore desideri disabilitare la bufferizzazione su di un file pointer può usare la funzione setbuf:

```
FILE *fp;
...
setbuf(fp, NULL);
```

dove fp è il file pointer, mentre il secondo parametro NULL richiede la disabilitazione completa della bufferizzazione.

13.5 Procedura anagrafica

Vediamo, a titolo di esempio, la gestione di un'anagrafica che abbiamo già esaminato nel Paragrafo 12.7 parlando di strutture dati composte tramite struttura, questa volta appoggiandoci su file in modo da rendere i dati persistenti (si veda il Listato 13.7), invece che lavorando in memoria tramite array di strutture.
Si osservi il modo in cui è gestito il file anag.dat. La funzione ins_per() apre il file:

```
fp = fopen("anag.dat","a+");
```

eventualmente creandolo, se non esiste, grazie all'opzione a+. Successivamente richiede all'utente i dati da inserire – cognome, nome, indirizzo ed età –, li memorizza nella struttura anag e provvede ad "appenderli" al file:

```
fwrite(&anag, sizeof(struct per), 1, fp);
```

La funzione di cancellazione eli_per(), invocata da can_per(), provvede a scrivere una struttura anag "vuota" su file, in corrispondenza della persona che deve essere eliminata. In questo caso l'apertura del file avviene con l'opzione r+:

```
fp = fopen("anag.dat","r+");
fseek(fp,pos,0);
```

```
fwrite(&anag, sizeof(struct per), 1, fp);
```

che consente di aggiornare il file a partire dalla posizione attuale del puntatore al file stabilita per mezzo della funzione di libreria `fseek()`.
La funzione `cer_per()`, invocata da `ric_per()`, effettua delle letture sul file per mezzo dell'istruzione

```
n = fread(&anag, sizeof(struct per), 1, fp);
```

alla ricerca della sequenza di byte che corrisponda a cognome, nome ed età della persona cercata.

```
#include <stdio.h>
#include <string.h>

#define DIM    31
#define MENU    0
#define INS     1
#define CAN     2
#define RIC     3
#define VIS     4
#define OUT   100

/* Semplice struttura che modella una persona */
struct per {
  char cognome[DIM];
  char nome[DIM];
  char ind[DIM];
  int  eta;
};

/* Puntatore al file */
FILE *fp;

/* La variabile di appoggio anag per le operazioni
sul file                                                    */
struct per anag;

int men_per(void);
void ins_per(void);
void ric_per(void);
void can_per(void);
long cer_per(char *, char *, int);
void eli_per(long pos);
void vis_per(void);
void vis_anagrafe(void);

/* Presenta il menu e lancia la funzione scelta */
void main()
{
int scelta = MENU;
while(scelta!=OUT) {
  switch(scelta) {
    case MENU:
      scelta = men_per();
      if(scelta == MENU)
         scelta = OUT;
      break;
    case INS:
      ins_per();
      scelta = MENU;
      break;
    case CAN:
```

```
      can_per();
      scelta = MENU;
      break;
    case RIC:
      ric_per();
      scelta = MENU;
      break;
  case VIS:
      vis_anagrafe();
      scelta = MENU;
      break;}
}
}

/* menu */
int men_per(void)
{
int scelta;
char invio;
int true = 1;

while(true){
  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
  printf("\t\t\t ANAGRAFE");
  printf("\n\n\n\t\t\t 1. Immissione Persona");
  printf("\n\n\t\t\t 2. Cancellazione Persona");
  printf("\n\n\t\t\t 3. Ricerca Persona");
  printf("\n\n\t\t\t 4. Visualizza anagrafe");
  printf("\n\n\t\t\t 0. Fine");
  printf("\n\n\n\t\t\t\t Scegliere una opzione: ");
  scanf("%d", &scelta);
  scanf("%c", &invio);
  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
  switch(scelta) {
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
      return(scelta);
    default:
      break;
  }
}
return(0);
}

/* inserisce persona nel file anag.dat */
void ins_per(void)
{
char invio;
/* Se il file non esiste lo crea,
   i dati sono appesi in fondo al file   */
fp = fopen("anag.dat","a+");

printf("\n\t\tINSERIMENTO PERSONA");
printf("\n\t\t---------------\n\n");
printf("\nCognome : ");
gets(anag.cognome);
printf("\nNome : ");
gets(anag.nome);
printf("\nIndirizzo : ");
gets(anag.ind);
```

```
printf("\nEtà : ");
scanf("%d", &anag.eta);
scanf("%c", &invio);

fwrite(&anag, sizeof(struct per), 1, fp);
fclose(fp);
}

/* Cancella una persona dall'anagrafe, se presente */
void can_per(void)
{
char pausa;
char cognome[DIM], nome[DIM];
int eta;
long pos;
printf("\n\t\tCANCELLA PERSONA");
printf("\n\t\t------\n\n");
printf("\nCognome : ");
gets(cognome);
printf("\nNome : ");
gets(nome);
printf("\nEtà : ");
scanf("%d", &eta);
scanf("%c", &pausa);
/* invoca ricerca persona */
pos = cer_per(cognome, nome, eta);
if(pos == -1) {
  printf("\nPersona non presente in anagrafe");
  scanf("%c", &pausa);
  return;
}
/* invoca visualizza persona */
vis_per();
printf("\nConfermi cancellazione ? (S/N) ");
scanf("%c", &pausa);
if(pausa=='S' || pausa=='s') {
  eli_per(pos);
  return;
}
}

/* Elimina persona dall'anagrafe */
void eli_per(long pos)
{
strcpy(anag.cognome, "");
strcpy(anag.nome, "");
strcpy(anag.ind, "");
anag.eta = 0;
fp = fopen("anag.dat","r+");
fseek(fp,pos,0);
fwrite(&anag, sizeof(struct per), 1, fp);
fclose(fp);
}

/* Ricerca persona */
void ric_per(void)
{
char pausa;
char cognome[DIM], nome[DIM];
int eta;
long pos;
/* Inserimento dati persona da ricercare */
printf("\n\t\tRICERCA PERSONA");
```

```
printf("\n\t\t------\n\n");
printf("\nCognome : ");
gets(cognome);
printf("\nNome : ");
gets(nome);
printf("\nEtà : ");
scanf("%d", &eta);
scanf("%c", &pausa);
/* Invoca la funzione di scansione sequenziale */
pos = cer_per(cognome, nome, eta);
if(pos == -1) {
  printf("\nPersona non presente in anagrafe");
  scanf("%c", &pausa);
  return;
}
vis_per();
scanf("%c", &pausa);
}

/* Effettua una scansione sequenziale del file anag.dat alla ricerca di una
persona che abbia determinati cognome, nome ed età */
long cer_per(char *cg, char *nm, int et)
{
int n;
long pos = 0L;

fp = fopen("anag.dat", "r");

for(;;) {
  n = fread(&anag, sizeof(struct per), 1, fp);
  if(n==0) {
    fclose(fp);
    pos = -1;
    return (pos);
  }
  else
    if(strcmp(cg, anag.cognome) == 0)
      if(strcmp(nm, anag.nome) == 0)
        if(et == anag.eta) {
            pos = ftell(fp);
              fclose(fp);
          return(pos-sizeof(struct per));
          }
  }
}

/* visualizza persona */
void vis_per(void)
{
printf("\n\n--------------------\n");
printf("\n\t\tCognome : %s", anag.cognome);
printf("\n\t\tNome : %s", anag.nome);
printf("\n\t\tIndirizzo : %s", anag.ind);
printf("\n\t\tEtà : %d", anag.eta);
printf("\n\n--------------------\n");
}

/* Visualizza l'anagrafe completa */
void vis_anagrafe(void)
{
int n; char pausa;
fp = fopen("anag.dat", "r");
do {
```

```
  n = fread(&anag, sizeof(struct per), 1, fp);
  if(n==0) fclose(fp);
  else {
    vis_per();
    scanf("%c", &pausa);
  }
}
while(n!=0);
}
```

Listato 13.7 Gestione anagrafica su file

13.6 Standard input e standard output

Quando un programma va in esecuzione il sistema apre automaticamente tre file pointer, mediante i quali è possibile scrivere messaggi a video e acquisire dati dalla tastiera. Questi tre file pointer prendono il nome di Standard Input (`stdin`), Standard Output (`stdout`) e Standard Error (`stderr`) e possono essere utilizzati dalle funzioni di accesso ai file descritte nei precedenti paragrafi.
Il file pointer `stdin` è associato dal sistema alla tastiera, i due file pointer `stdout` e `stderr` sono entrambi assegnati al video. Per scrivere un messaggio a video si può allora utilizzare, per esempio, la funzione `fprintf`:

```
#include <stdio.h>
main()
{
fprintf(stdout,"Ciao lettore\n");
}
```

Dunque le funzioni che abbiamo utilizzato per accettare valori da tastiera e mandare uscite al video corrispondono a usi particolari delle funzioni di uso generale esaminate in questo capitolo. Si hanno le seguenti equivalenze:

```
printf(...)    ->    fprintf(stdout,...)
scanf(...)     ->    fscanf(stdin,...)
getchar()      ->    fgetc(stdin)
putchar(c)     ->    fputc(c, stdout)
eof()          ->    feof(stdin)
```

La sintassi delle funzioni a sinistra è più sintetica, perché quelle sulla destra devono specificare che desiderano operare sullo standard input o sullo standard output.
Il programmatore deve prestare molta attenzione all'utilizzo delle due funzioni con sintassi abbreviata `gets` e `puts`, il cui comportamento è simile ma non uguale a quello delle funzioni `fgets` e `fputs`. Infatti `gets` legge una riga da tastiera ma elimina il carattere di newline e `puts` scrive una riga a video aggiungendo automaticamente un carattere di newline. Ne consegue che :

`gets(buf,n)` non equivale a `fgets(buf, n, stdin)`
`puts(buf)` non equivale a `fputs(buf, stdout)`

13.7 Funzioni di basso livello

Alcune versioni del C offrono un altro insieme di funzioni per operare sui file. La modalità d'uso di queste funzioni è assai simile a quella delle funzioni precedenti, ma il loro impiego è limitato allo sviluppo di applicazioni che abbiano la necessità di raggiungere notevoli prestazioni. Queste funzioni sono di più basso livello rispetto alle precedenti e corrispondono spesso direttamente alle chiamate al sistema operativo. Il lettore, se lo desidera, può saltare questo

paragrafo a una prima lettura.
Il programmatore deve prestare molta attenzione a non utilizzare contemporaneamente le due classi di funzioni sul medesimo file; infatti le due strategie interne di gestione dei file sono differenti e l'uso contemporaneo delle due classi di funzioni può generare errori ed effetti collaterali all'interno del programma la cui portata non è valutabile a priori.
Le funzioni precedenti utilizzavano il concetto di file pointer per operare sui file, quelle che andiamo a descrivere ora implementano un concetto analogo, che prende il nome di *file descriptor*, talvolta chiamato "maniglia" o "canale". Il file descriptor è un numero intero che viene associato dalla funzione di apertura al file sul quale si desidera operare.
Per lavorare con queste funzioni di accesso ai file è necessario includere: `fcntl.h`, `sys/types.h` e `sys/stat.h`.
La funzione per aprire un file si chiama `open` e lavora come nell'esempio seguente:

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

main()
{
int fd;
fd = open("clienti", O_RDONLY);
...
```

Tale funzione associa il file descriptor `fd` al file di nome `clienti` aprendolo in modalità di sola lettura. Il valore di ritorno della `open` può essere negativo nel caso in cui si sia verificato un errore, per esempio se il file `clienti` non esiste. L'uso generale della funzione `open` su `clienti` è:

```
int fd, modo, diritti;
...
fd = open("clienti", modo [diritti] );
```

dove *modo* rappresenta la modalità di apertura del file è può avere il valore di una o più delle seguenti costanti simboliche:

O_RDONLY	apre il file in sola lettura; se il file non esiste la `open` ritorna errore
O_WRONLY	apre in file in sola scrittura; se il file non esiste la `open` ritorna errore
O_RDWR	apre il file in lettura e scrittura; se il file non esiste la `open` ritorna errore
O_CREAT	crea il file se non esiste
O_TRUNC	distrugge il contenuto del file preesistente
O_APPEND	apre il file in modalità append; tutte le scritture sono automaticamente eseguite a fine file
O_EXCL	la `open` ritorna errore se il file esiste già al momento dell'apertura

Se il programmatore desidera specificare più di una modalità di apertura lo può fare utilizzando l'operatore binario di OR bit a bit; per esempio

```
fd = open("clienti", O_WRONLY | O_TRUNC);
```

fornisce l'apertura del file `clienti` in scrittura con distruzione del contenuto preesistente.

Nel caso in cui sia stata specificata la modalità `O_CREAT` il programmatore deve anche specificare i `diritti` o permessi con i quali il file deve essere creato; tali permessi sono solitamente codificati con una sintassi simile a quella utilizzata dal sistema operativo Unix, in cui chi utilizza un file rientra sempre in almeno una di queste categorie:

- • è il possessore del file;
- • appartiene al gruppo collegato al file;
- • non è collegato in alcun modo al file.

Per ciascuna categoria, è possibile specificare i permessi che consentono l'utilizzazione del file mediante la forma ottale, che è costituita da tre o quattro cifre comprese tra 0 e 7, per esempio:

`0640`

In questo contesto tralasciamo il significato della prima cifra a sinistra che è opzionale. Ogni cifra viene interpretata come una somma delle prime 3 potenze di 2 ($2^0=1$, $2^1=2$, $2^2=4$), ciascuna delle quali corrisponde a un determinato tipo di permesso – dove la seconda rappresenta il proprietario, la terza il gruppo e l'ultima a destra tutti gli altri utenti –; la corrispondenza è la seguente:

`4` permesso di lettura
`2` permesso di scrittura
`1` permesso di esecuzione
`0` nessun permesso

Dunque con `640` abbiamo i permessi di lettura e scrittura per il proprietario (`4` permesso di lettura + `2` permesso di scrittura), di sola lettura per il gruppo e nessun permesso agli altri.

Vediamo un esempio dove definiamo più di una modalità e anche i `diritti`: il codice

```
int fd;
fd = open("clienti", O_RDWR | O_CREAT | O_TRUNC, 0640);
```

crea il file `clienti` aprendolo in lettura e scrittura con diritti di lettura/scrittura per il proprietario (`6`), con soltanto diritto di lettura per il gruppo di utenti cui appartiene il proprietario (`4`) e con nessun diritto per tutti gli altri utenti (`0`).
La funzione `close` ha un comportamento analogo a quello della funzione `fclose`: chiude un file descriptor aperto dalla `open`:

```
int fd;
...
close(fd);
```

Per quanto riguarda le operazioni di lettura e scrittura su file con utilizzo dei file descriptor, le funzioni che le eseguono si chiamano rispettivamente `read` e `write`.
La funzione `read` opera nel modo seguente:

```
char buf[1000];
int elementi;
int fd;
int n;
...
elementi = 1000;
n = read(fd, buf, elementi);
```

dove `fd` è il file descriptor da cui si desidera leggere, `buf` è il vettore dove i dati letti devono essere trasferiti ed `elementi` rappresenta la dimensione in byte del vettore. Il valore di ritorno della funzione indica quanti byte sono stati realmente letti dal file `fd`; tale valore può essere inferiore alla dimensione di `buf` nel caso in cui il puntatore al file abbia raggiunto la fine; un valore di ritorno uguale a zero indica che siamo giunti a fine file.
La funzione `fwrite` lavora così:

```
char buf[1000];
int elementi = 1000;
int n, fd;
...
n = write(fd, buf, elementi);
```

dove `fd` è il file dove si desidera scrivere, `buf` contiene i dati che devono essere scritti ed `elementi` rappresenta il numero di byte da scrivere. Il valore di ritorno della `write` indica il numero di byte scritti sul file; tale valore può essere inferiore a `elementi` nel caso in cui il file abbia superato la massima dimensione ammessa.
Scriviamo un programma di esempio che copia il contenuto di un file in un altro utilizzando i file descriptor (Listato 13.8).

✓ **NOTA**

Questa volta abbiamo utilizzato la funzione `exit` che fa terminare il programma (si veda il Paragrafo 3.8). I programmatori C usano spesso questa possibilità del linguaggio, che noi abbiamo circoscritto agli esempi di questo paragrafo. Se non si ritiene opportuno questo approccio è sufficiente togliere la `exit` e aggiungere un ramo `else` a `if(fp==NULL)`, con un blocco dove raccogliere la parte seguente del programma.

```
/* Copia il contenuto di un file in un altro */

#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

main(argc,argv)
int argc;
char **argv;
{
static char buf[BUFSIZ];
int fdin, fdout, n;

if( argc != 3 ) {
   printf("Devi specificare file sorgente e destinazione\n");
   exit(1);
}

/* Apre il file sorgente */
fdin = open(argv[1],O_RDONLY);
if( fdin < 0 ) {
   printf("Non posso aprire %s\n",argv[1]);
   exit(2);
}

/* Apre il file destinazione */
fdout = open(argv[2],O_WRONLY|O_CREAT|O_TRUNC,0600);
if( fdout < 0 ) {
   printf("Non posso creare %s\n", argv[2]);
   close(fdin);
   exit(3);
}

/* Esegue il ciclo di lettura e scrittura */
for(;;) {
   /* Legge BUFSIZ caratteri da file */
   n = read(fdin, buf, BUFSIZ);

   /* Controlla la fine del file */
   if( n == 0 )
      break;

   /* Scrive i caratteri nel file destinazione */
   write(fdout,buf,n);
}

/* Chiude i file */
close(fdin);
close(fdout);
```

```
}
```

Listato 13.8 Programma che copia il contenuto di un file in un altro utilizzando i file descriptor

Abbiamo visto in precedenza come sia possibile spostare il file pointer utilizzando la funzione `fseek`; anche per i file descriptor esiste una funzione, `lseek`, che consente di muovere il puntatore al file:

```
long offset;
long n;
int mode;
int fd;
...
offset = lseek(fd, n, mode);
```

dove `fd` è il file descriptor sul quale si desidera muovere il puntatore, `n` rappresenta il numero di byte di spostamento. Se `n` è negativo lo spostamento del puntatore avviene all'indietro invece che in avanti. Il parametro `mode` indica a partire da quale posizione iniziare a muovere il puntatore: se `mode` vale `0` significa che ci si deve muovere a partire dall'inizio del file, se vale `1` a partire dalla posizione corrente e infine se vale `2` a partire dalla fine. Il valore di ritorno della `lseek` contiene la posizione corrente del puntatore dopo lo spostamento. Allora:

`lseek(fd,0L,1)`	restituisce la posizione corrente
`lseek(fd,0L,2)`	restituisce la dimensione del file in byte

Utilizzando i file pointer, avevamo visto come il sistema offriva tre file pointer automaticamente aperti al momento del lancio del programma (`stdin`, `stdout`, `stderr`) mediante i quali era possibile lavorare su tastiera e video. Anche usando i file descriptor il sistema mette a disposizione tre descrittori aperti per default al momento del lancio del programma e associati a tastiera e video:

`0`	standard input associato a tastiera
`1`	standard output associato al video
`2`	standard error associato al video

Questi tre numeri interi possono essere utilizzati dal programmatore per leggere e scrivere da tastiera e video senza dover eseguire le relative `open`. Scriviamo dunque un programma di esempio che memorizza all'interno di un file informazioni su un gruppo di alunni inserite da tastiera (Listato 13.9).

```
/* Memorizza in un file le informazioni passate dall'utente sugli alunni di un
classe */

#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

/* La struttura alunno contiene nome, cognome */
/* ed età di ogni alunno                     */
struct alunno {
   char nome[100];
   char cognome[100];
   int eta;
};


main()
{
struct alunno alunno;
int nalunni;
int fd;

/* Apre il file alunni */
```

```
fd = open("alunni",O_WRONLY|O_CREAT|O_TRUNC,0600);
if( fd < 0 ) {
   printf("Non posso aprire il file alunni\n");
   exit(1);
}

printf("Quanti alunni vuoi inserire ? ");
scanf("%d",&nalunni);

while( nalunni-- > 0 ) {
   printf("Nome : ");
   scanf("%s",alunno.nome);
   printf("Cognome : ");
   scanf("%s",alunno.cognome);
   printf("Età: ");
   scanf("%d",&alunno.eta);
   write(fd, &alunno, sizeof(struct alunno));
}
close(fd);
}
```

Listato 13.9 Programma che memorizza all'interno di un file informazioni su un gruppo di alunni inserite da tastiera

13.8 Embedded SQL

I programmi C possono interagire con i sistemi di gestione di basi di dati attraverso il linguaggio SQL (*Structured Query Language*), in modalità *embedded* (letteralmente: "incastrato"), cioè accogliendo al loro interno pezzi di codice SQL, passando dei parametri e ricevendo dei risultati.

SQL è il linguaggio standard, utilizzato dai DBMS relazionali (RDBMS), che consente l'esecuzione di tutte le operazioni necessarie nella gestione e nell'utilizzo di una base di dati; permette infatti, oltre alla consultazione del database, anche la definizione e gestione dello schema, la manipolazione dei dati e l'esecuzione di operazioni amministrative.

> ✓ **NOTA**
> Con il termine database (base di dati) si indica un insieme di dati rivolti alla rappresentazione di uno specifico sistema informativo di tipo aziendale, scientifico, amministrativo o altro, mentre con sistema di gestione di base di dati o Data Base Management System (DBMS) ci si riferisce a un componente di software di base che consente la gestione del database. I DBMS hanno permesso di superare l'approccio tradizionale al problema dell'archiviazione: l'utilizzo diretto dei file sulle strutture del file system, in cui le applicazioni accedono direttamente agli archivi dei dati – come nei programmi visti fino a qui in questo capitolo – , in cui ogni applicazione deve conoscere la struttura interna degli archivi e le relazioni tra i dati, deve preoccuparsi che siano rispettati i requisiti di ridondanza minima. Nell'approccio tradizionale rimangono poi da soddisfare i requisiti di utilizzo contemporaneo degli archivi da parte di più applicazioni e di permanenza dei dati, che spesso vengono in parte delegati a strati sottostanti di software non specializzato quali il sistema operativo.
> Rispetto alle soluzioni tradizionali, l'utilizzo di un DBMS comporta una serie di vantaggi che si traducono in una gestione dei dati più affidabile e coerente. In particolare ne derivano: indipendenza dei dati dall'applicazione, riservatezza nell'accesso ai dati, gestione dell'integrità fisica dei dati, gestione dell'integrità logica dei dati, sicurezza e ottimizzazione nell'uso dei dati (si veda la Figura 13.1).

Perché l'interazione tra C e sistema di gestione di una base di dati sia possibile deve essere disponibile un apposito precompilatore – generalmente messo a disposizione dalla casa madre del database in oggetto – che preprocessa il programma applicativo e produce un programma C che può essere poi normalmente compilato (Figura 13.2). Per esempio, il precompilatore della Oracle è conosciuto come Pro-C mentre quello della Sybase come Embedded SQL/C. In qualsiasi parte del programma possono essere inseriti comandi SQL immersi all'interno del comune codice C; nel caso di Sybase i comandi devono essere preceduti dalle parole chiave `exec sql` e terminati da un punto e virgola (Listato 13.10).

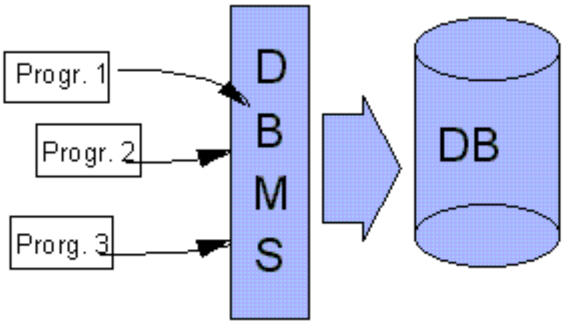


Figura 13.1 I programmi accedono ai dati attraverso il gestore di basi di dati

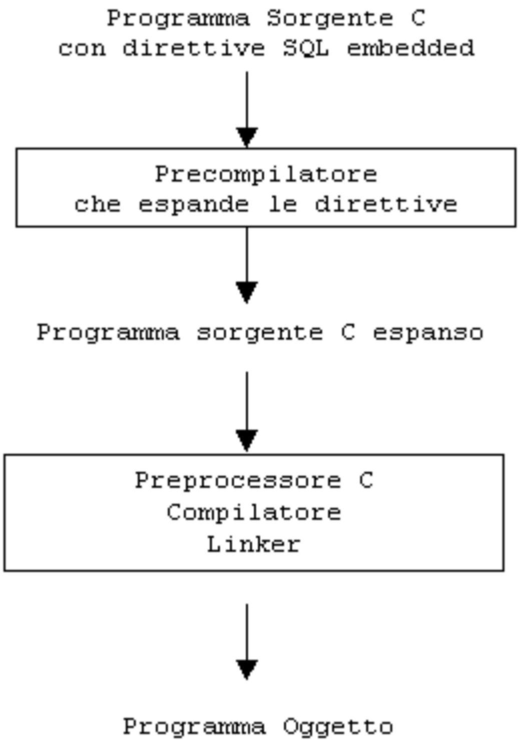


Figura 13.2 La prima fase dell'interazione fra C e DBMS è l'espansione delle direttive Embedded SQL

```
/* Esempio di Embedded SQL */

exec sql include sqlca;

main()
{
/* Dichiarazione delle variabili */
exec sql begin declare section;
CS_CHAR utente[31]; password[31];
exec sql end declare section;

/* Inizializzazione del sottoprogramma che
   gestisce gli errori SQL                  */
exec sql whenever sqlerror perform err_p();

/* Avvia la connessione con il server SQL */
printf("\nInserisca l'identificativo utente ");
gets(utente);
printf("\npassword ");
gets(password);
exec sql connect :user identified by :password;

/* Esempio di esecuzione di un comando SQL */
exec sql update auto set prezzo = prezzo *1.10;

/* Chiude la connessione con il server SQL */
exec sql disconnect;
}

/* Sottoprogramma che gestisce gli errori SQL */
```

```
err_p()
{
/* Stampa il codice di errore, un messaggio e il
   numero di linea dove si è prodotto l'errore */
printf("\nerrore nella linea %d, codice errore %d.\ns"
        sqlca.sqlcode, sqlca.sqlerrm, sqlerrmc);
}
```

Listato 13.10 Memorizzazione all'interno di un file di informazioni su un gruppo di alunni, inserite da tastiera

Nel caso di Sybase – per gli altri database la sintassi si discosta un po' ma i concetti rimango validi – ogni direttiva inserita nel programma C ha la forma

exec sql *direttivaSQL*

Nel programma del Listato 13.10 inizialmente viene incluso il file `sqlca` con la direttiva

exec sql include sqlca;

in modo analogo a come facciamo con la direttiva `#include` del preprocessore C. Il file `sqlca` contiene strutture dati necessarie al sistema per eseguire i comandi SQL. Successivamente vengono dichiarate le variabili `utente` e `password` di tipo array di `CS_CHAR`, che è equivalente in C a un array di `char`, dunque a una stringa.

```
exec sql begin declare section;
CS_CHAR utente[31]; password[31];
exec sql end declare section;
```

Il programma richiede l'identificativo utente e la password e la comunica al database con altra direttiva:

```
exec sql connect :user identified by :password;
```

Per ragioni di sicurezza, infatti, si inizia normalmente una sessione di lavoro facendosi riconoscere dal gestore di basi di dati. Ed ecco che finalmente eseguiamo un'operazione direttamente sulla base di dati con un comando SQL:

```
exec sql update auto set prezzo = prezzo *1.10;
```

Il comando `update` (aggiorna) modifica la tabella `auto` aumentando il valore della colonna `prezzo` del 10%. L'ultima direttiva chiude la connessione con il database:

```
exec sql disconnect;
```

Naturalmente tra l'apertura e la chiusura della sessione di lavoro avremmo potuto inserire a piacere codice C e altre direttive Embedded SQL ■.

13.9 Esercizi

* 1. Scrivere un programma che legga e visualizzi il contenuto di un file ASCII, per esempio `autoexec.bat`.

* 2. Scrivere un programma che apra un file e vi inserisca 80 caratteri.

* 3. Scrivere un programma che offra tramite menu le funzioni di inserimento, ricerca e cancellazione di un archivio `studenti`.

* 4. Scrivere un programma che apra un file, legga e visualizzi una riga, poi torni indietro all'inizio del file e legga nuovamente una linea. Ovviamente le due letture devono produrre il medesimo risultato.

* 5. Aprire un file ASCII, per esempio `autoexec.bat`, e leggere e visualizzare i primi 10 gruppi di caratteri separati da blank e newline.

6. Scrivere un programma che permetta di gestire una rubrica telefonica in modo che i dati vengano memorizzati in forma permanente sul file `rubrica`. Offrire all'utente un menu con le opzioni: inserimento, modifica, cancellazione e visualizzazione dell'intera rubrica.

7. Scrivere una funzione che, dato in ingresso il nome, cerchi in `rubrica` il corrispondente numero di telefono. Aggiungere al menu del programma dell'Esercizio 6 l'opzione che richiama tale procedura.

8. Scrivere una funzione che permetta l'ordinamento del file `rubrica` rispetto al nome. Aggiungere al menu del programma dell'Esercizio 6 l'opzione che richiama tale procedura.

9. Scrivere una funzione che permetta, una volta che il file `rubrica` è ordinato per nome, di effettuare una ricerca binaria. Aggiungere al menu del programma dell'Esercizio 6 l'opzione che richiama tale procedura.

10. Scrivere una procedura che visualizzi tutti i dati delle persone del file `rubrica` i cui nomi iniziano con una lettera richiesta all'utente. Aggiungere al menu del programma dell'Esercizio 6 l'opzione che richiama tale procedura.

11. Modificare il programma dell'Esercizio 6 in modo tale che il file `rubrica` contenga, oltre al nome e al numero di telefono, anche il cognome e l'indirizzo (via, CAP, città e stato) dei conoscenti memorizzati.

12. Un'azienda vuole memorizzare nel file `dipendenti` i dati relativi a ogni dipendente. In particolare si vogliono archiviare nome, cognome, sesso (M,F), anno di nascita e città di residenza. Scrivere un programma che crei un tale file e memorizzi i dati relativi ad alcuni dipendenti.

13. Rispetto al file dell'esercizio precedente scrivere le funzioni per stampare:

1. 1. la lista (nome e cognome) dei dipendenti che hanno più di 50 anni;
2. 2. la lista (nome e cognome) dei dipendenti residenti a Viterbo;
3. 3. il totale dei dipendenti maschi, quello dei dipendenti femmine e il numero complessivo dei dipendenti.

14. Una biblioteca memorizza i libri richiesti in lettura in un file in cui sono riportati nome e cognome del richiedente, autore e titolo del libro e il codice dell'argomento compreso fra 1 e 25. Scrivere un programma che crei un tale file e memorizzi i dati relativi ad alcuni libri in prestito. Realizzare le funzioni necessarie alla gestione del file e che permettano alla fine del mese di stampare: tutti i record del file, il numero totale dei lettori, la lista dei libri che sono stati richiesti per ognuno dei 25 argomenti e il loro numero totale, quindi l'argomento con il maggior numero di libri richiesti. Successivamente si provi a realizzare una soluzione migliore memorizzando libri, autori e argomenti in file distinti, relazionati mediante codici univoci appunto di libro, autore, argomento.

15. Un'associazione culturale memorizza nel file `sovvenzioni` la sede che ha ottenuto la sovvenzione, il nome del socio che ha effettuato il versamento nonché la data e l'importo del versamento stesso. Scrivere un programma che crei un tale file e memorizzi i dati relativi ad alcune sovvenzioni e le funzioni che determinano: la sede che ha ottenuto la sovvenzione più grande, la sede che ha il totale complessivo dei finanziamenti più grande, il totale delle sovvenzioni ottenute nel dicembre 1997, se il 26.11.95 il socio Marco Taddei ha effettuato un versamento (e in caso positivo ne

visualizzi tutti i dati), la lista di tutti i finanziamenti ottenuti dalla sede di Vercelli. Anche in questo caso, come nell'esercizio precedente, si cerchi successivamente una soluzione migliore per la memorizzazione delle informazioni e per la loro gestione.

16. [*Fusione di due file*] Supponendo che i file `rubrica1` e `rubrica2` siano ordinati per nome, scrivere un programma che crei un nuovo file ordinato `rubrica` che contiene i dati di entrambi i file.

17. Modificare a scelta alcuni dei programmi visti nei capitoli precedenti in modo da lavorare, invece che su vettori in memoria centrale, su file in memoria di massa.

14.1 Limite degli array

Le strutture dati individuate per la risoluzione dei problemi sono dette "astratte". Una *struttura dati astratta* definisce l'organizzazione delle informazioni indipendentemente dalla traduzione in uno specifico linguaggio di programmazione; la rappresentazione che una struttura astratta prende in un linguaggio di programmazione viene detta *implementazione*.
Le strutture astratte dipendono dal problema in considerazione; alcune di esse, come le pile, le code e gli alberi, rivestono però un'importanza generale per l'uso comune che se ne fa nella soluzione di intere classi di problemi. Nella fase di analisi il programmatore si può accorgere che proprio una di queste strutture fa al caso suo; successivamente, dovrà scegliere la tecnica migliore per implementarle. Per facilitare il lavoro del programmatore, il C mette a disposizione una serie di strumenti, tra i quali gli array, le `struct` e i puntatori. In questo capitolo e nel seguente faremo pratica di programmazione implementando alcune tra le più importanti strutture dati.
Fino qui, i dati omogenei sono stati memorizzati con l'array. Questi ultimi, però, possono porre dei problemi, in merito a:

- • occupazione di memoria;
- • velocità di esecuzione;
- • chiarezza della soluzione proposta.

Consideriamo in primo luogo l'*occupazione di memoria*. Il numero di elementi di un array viene definito nelle parti dichiarative del programma. È essenziale dunque, in fase di analisi del problema, effettuare una stima sul massimo numero di elementi che eventualmente potranno essere utilizzati. In molti casi non è possibile dare una valutazione esatta, per cui si dovrà sovrastimare l'array, cioè definirlo con un numero di elementi probabilmente molto superiore al necessario, occupando più memoria di quella realmente indispensabile.

Per esempio, si supponga di elaborare le prenotazioni di 700 studenti delle scuole medie relativamente a una delle tre gite scolastiche previste nell'anno: la prima a Vienna, la seconda a Parigi e la terza a Roma. Per formare le tre liste di studenti si possono utilizzare tre array di caratteri: `Vienna`, `Parigi` e `Roma`. Ognuno di essi dovrà essere definito con 700 elementi, perché non si conosce a priori la distribuzione delle scelte degli studenti, per cui si deve considerare il caso peggiore, in cui tutti gli studenti scelgono la stessa gita. In questo caso si avrà una occupazione di memoria pari (o superiore, se qualche studente non partecipa a nessuna gita) a tre volte quella effettivamente richiesta.
Consideriamo ora il secondo fattore: la *velocità di esecuzione*. Il fatto che gli elementi di un array siano rigidamente connessi l'uno all'altro in una successione lineare appesantisce gli algoritmi di risoluzione di alcuni problemi, rendendo non accettabili i tempi di risposta dei programmi corrispondenti.
Si supponga per esempio di dover elaborare una lista ordinata di interi e prevedere le operazioni di inserimento ed eliminazione su questa lista. Se si memorizza la lista in un array a una dimensione l'operazione di inserimento di un valore deve prevedere le fasi:

1. 1. ricerca del punto d'inserimento;
2. 2. spostamento di un posto di tutti gli elementi successivi;
3. 3. inserimento dell'informazione nell'array.

Si osservi al proposito l'esempio di Figura 14.1, dove in una lista costituita dai valori 3, 7, 20, 31, 100, 200 memorizzata nei primi sei elementi di un vettore vogliamo inserire il valore 11 mantenendo l'ordinamento. In questa sede è interessante sottolineare la fase di spostamento in avanti dei valori che seguono l'11, fase in cui sono necessari ben quattro spostamenti prima di poter inserire il nuovo elemento effettuando in pratica quattro riscritture.

Poiché in generale il numero di spostamenti corrisponde al numero degli elementi del vettore successivi a quello da inserire, per ogni inserimento in un vettore che già contiene *n* valori dovremo in media effettuare *n*/2 riscritture. Analoghe considerazioni valgono per una cancellazione.

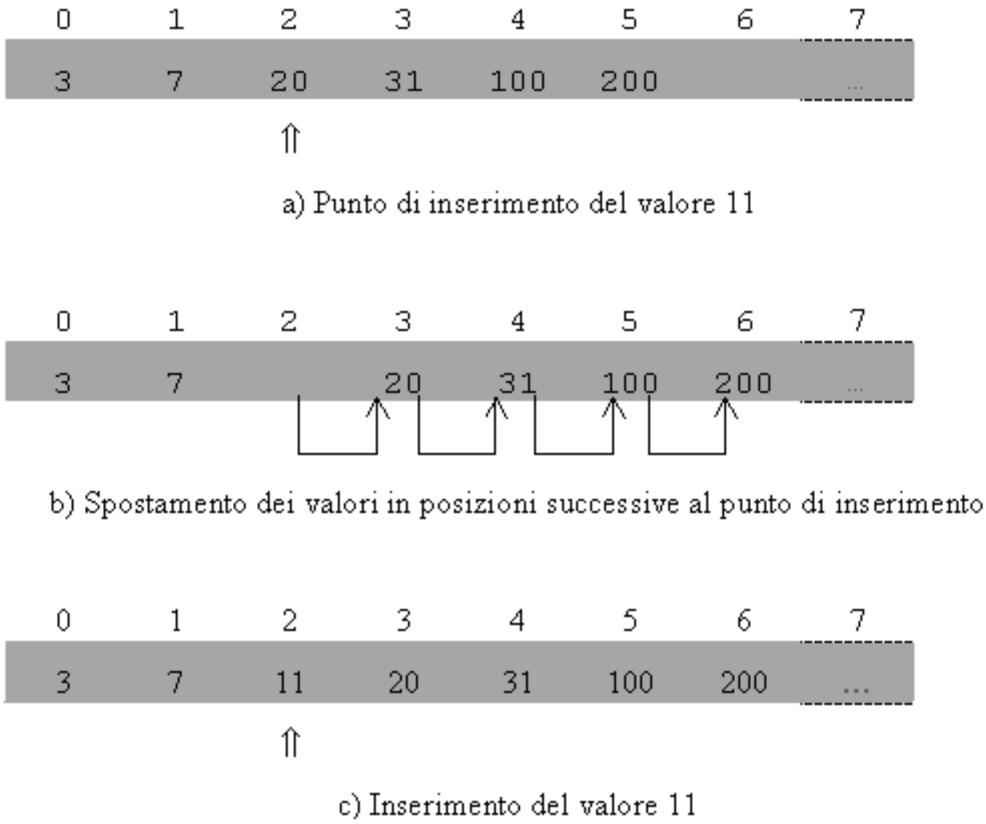


Figura 14.1 Inserimento ordinato di un elemento nella sequenza memorizzata in un array.

Consideriamo infine il terzo fattore: la *chiarezza della soluzione proposta*. Più l'organizzazione della struttura corrisponde alla logica di utilizzazione delle informazioni in essa contenute, più è chiara la soluzione. Questo fattore ha un riflesso determinante sulla bontà della programmazione, in termini di diminuzione dei tempi di revisione e modifica dei programmi. In sintesi, i limiti che vengono ravvisati nell'utilizzazione degli array si collegano alla poca elasticità/flessibilità insita in questa struttura. Non sempre l'array corrisponde alla scelta migliore.

14.2 Liste lineari

Una lista lineare è una successione di elementi omogenei che occupano in memoria una posizione qualsiasi. Ciascun elemento contiene un'informazione e un puntatore per mezzo del quale è legato al successivo. L'accesso alla lista avviene con il puntatore al primo elemento. Si ha dunque:

```
Elemento = informazione + puntatore
```

Il puntatore è il riferimento a un elemento, il suo valore è l'indirizzo dell'elemento nella memoria del sistema. Indichiamo con `inf` la parte informazione di ogni elemento, con `pun` la parte puntatore e con `punt_lista` il puntatore al primo elemento della lista, come in Figura 14.2.

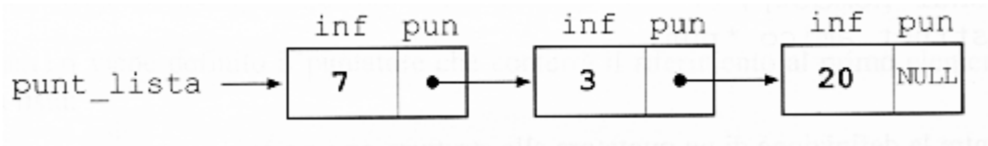


Figura 14.2 Esempio di lista lineare

Il campo puntatore dell'ultimo elemento della lista non fa riferimento a nessun altro elemento; il suo contenuto corrisponde a un segnale di fine lista che in C è il valore `NULL`. Una *lista vuota* non ha elementi ed è rappresentata da `punt_lista` che punta a `NULL`.

`punt_lista` ⟶ `NULL`

La parte informazione dell'elemento dipende dal tipo di dati che stiamo trattando. In C può essere costituita da uno qualsiasi dei tipi semplici che conosciamo: `int`, `float` ecc. Nel caso della lista lineare di Figura 14.2 il cui campo informazione è di tipo intero, la dichiarazione della struttura di ogni elemento può essere la seguente:

```
struct elemento {
       int inf;
       struct elemento *pun;
     };
```

dove `inf` è di tipo `int` e `pun` è un puntatore a una struttura di tipo elemento. La scelta del nome della struttura e dei nomi dei campi, in questo caso elemento, `inf` e `pun`, è libera, nell'ambito degli identificatori ammessi dal linguaggio. La precedente dichiarazione descrive la struttura di elemento ma non alloca spazio in memoria. La definizione:

```
struct elemento *punt_lista;
```

stabilisce che `punt_lista` è un puntatore che può riferirsi a variabili `elemento`. Non esistono variabili puntatore in generale, ma variabili puntatore che fanno riferimento a oggetti di un determinato tipo.

La parte informazione può essere anche composta da più campi, ognuno dei quali di un certo tipo. Per esempio, se si desiderano memorizzare nella lista nomi ed età degli amici la parte informazione diventa:

```
informazione = nome + anni
```

con nome di tipo array di `char` e `anni` di tipo `int`. La dichiarazione di un elemento diventa allora:

```
struct amico {
       int anno;
       char nome[30];
       struct amico *pun;
     };
```

mentre la definizione di un puntatore alla struttura amico è:

```
struct amico *p_amico;
```

È da sottolineare la posizione in memoria non sequenziale: quando si aggiunge un ulteriore elemento alla lista si deve allocare uno spazio di memoria, connetterlo all'ultimo elemento della lista e inserirvi l'informazione relativa.
Nella struttura lista lineare, così come definita, non esiste alcun modo di risalire da un elemento al suo antecedente. La lista si può *scandire* solo in ordine, da un elemento al successivo, per mezzo dei puntatori. *Scandire* una lista significa esaminare uno per uno i suoi elementi, dove per esaminare si può intendere: leggere, stampare ecc. Il segnale di fine lista `NULL` è importante perché permette di verificare durante la scansione se la lista è terminata.
I problemi che vengono presentati e risolti in questo capitolo permettono di familiarizzare con le liste lineari.

14.3 Gestione di una lista

Consideriamo il problema di memorizzare e successivamente visualizzare una sequenza di *n* interi. Il valore di *n* non è conosciuto a priori, ma è determinato in fase di esecuzione. Se si decide di utilizzare la struttura informativa array si deve prefissare il numero massimo di valori della sequenza. Si propende quindi per una soluzione che utilizzi la memoria in modo dinamico. Il tipo `elemento` è una struttura composta da due campi, un campo `inf` di tipo `int` e un campo `pun` puntatore alla struttura stessa:

```
struct elemento {
       int inf;
       struct elemento *pun;
     };
```

Il problema presentato è divisibile nei due sottoproblemi:

- memorizzare la sequenza;
- visualizzare la sequenza.

Si affida quindi la soluzione dei due sottoproblemi a due funzioni, la cui dichiarazione prototype è:

```
struct elemento *crea_lista();
     void visualizza_lista(struct elemento *);
```

Nel `main` viene definito il puntatore che conterrà il riferimento al primo elemento della lista:

```
struct elemento *punt_lista;
```

Le due funzioni vengono chiamate in sequenza dallo stesso `main`; il programma completo è riportato nel Listato 14.1.

```
punt_lista = crea_lista();
     visualizza_lista(punt_lista);
```

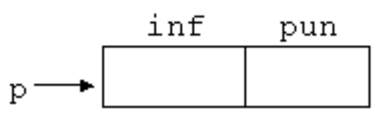
La procedura `crea_lista` restituisce al `main` il puntatore alla lista che è assegnato a `punt_lista` e che viene successivamente passato a `visualizza_lista`.
La funzione `crea_lista` è di tipo puntatore a strutture `elemento`. Non prevede il passaggio di valori dal chiamante, per cui una sua dichiarazione più appropriata sarebbe stata:

```
struct elemento *crea_lista(void);
```

dove `void` esplicita la mancanza di parametri. Nel nostro caso, dunque, il compilatore potrebbe segnalare un *warning* (attenzione!).
La funzione `crea_lista` deve comprendere la dichiarazione di `p`, puntatore alla testa della lista, e di `paus`, puntatore ausiliario, che permette la creazione degli elementi successivi al primo, senza perdere il puntatore iniziale alla lista. In primo luogo si deve richiedere all'utente di inserire il numero di elementi da cui è composta la lista. Questa informazione viene memorizzata nella variabile `n`, di tipo `int`. Dopo di ciò, se `n` è uguale a zero, si assegna a `p` il valore `NULL`, che corrisponde a lista vuota. In questo caso il sottoprogramma termina. Si osservi che `n` è una variabile locale alla funzione.
Se il numero di elementi (valore di `n`) è maggiore di zero `crea_lista` deve creare il *primo elemento*:

```
p = (struct elemento *)malloc(sizeof(struct elemento));
```

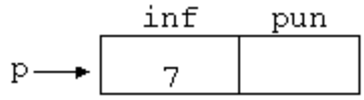
L'operatore `sizeof` restituisce la quantità di memoria occupata da un elemento e `malloc` alloca uno spazio corrispondente di memoria libera. Successivamente viene effettuato un *cast* del valore ritornato da `malloc`, in modo da trasformarlo in un puntatore allo spazio allocato che viene assegnato a `p`.



Per richiamare `malloc` si deve includere nel programma il riferimento alla libreria `malloc.h` e/o `stdlib.h`; in implementazioni del C meno recenti la libreria da includere è stddef.h. Si richiede all'utente di inserire la prima informazione che viene assegnata `p->inf`, campo `inf` del primo elemento della lista:

```
scanf("%d", &p->inf);
```

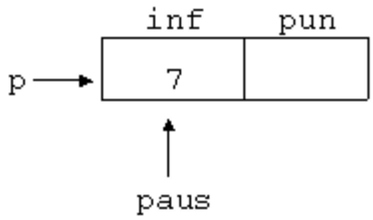
Si supponga che l'utente abbia passato precedentemente il valore 3 come lunghezza della sequenza e successivamente immetta i seguenti valori: 7, 3, 20; dopo la prima immissione avremo la seguente situazione:



dopo di che viene assegnato a `paus` il valore di `p`:

```
paus = p;
```

Come per ogni altro tipo di variabile semplice, è possibile fare assegnamenti fra puntatori purché si riferiscano allo stesso tipo di oggetto. Da questo momento sia `p` sia `paus` (puntatore ausiliario) fanno riferimento al primo elemento della lista.



Finita la gestione del primo elemento, deve iniziare un ciclo per la creazione degli *elementi successivi al primo*. Questo ciclo si ripete *n*–1 volte, dove *n* è la lunghezza della sequenza in ingresso. Il ciclo è così costituito:

`for(i = 2; i<=n; i++) {`
a) *crea il nuovo elemento concatenato al precedente*
b) *sposta di una posizione avanti* paus
c) *chiedi all'utente la nuova informazione della sequenza*; *inserisci l'informazione nel campo* `inf` *del nuovo elemento*
`}`

Nel caso dell'esempio proposto, il ciclo si ripete due volte (da `i`=2 a `i`=`n`=3); a ogni iterazione le operazioni descritte corrispondono a:

```
a) paus->pun = (struct elemento *)malloc(sizeof(struct elemento));
b) paus = paus->pun;
c) printf("\nInserisci la %d informazione: ", i);
   scanf("%d", &paus->inf);
```

In a) viene creato un nuovo elemento di lista, connesso al precedente. In questo caso il puntatore ritornato dall'operazione di allocazione di memoria e successivo *cast* viene assegnato al campo puntatore della variabile puntata da `paus`. Infatti `paus->` specifica che si tratta dell'elemento puntato da `paus` e `pun` indica il campo della struttura cui si fa riferimento (Figura 14.3a). In b) viene attualizzato il puntatore ausiliario `paus`, in modo da farlo puntare all'elemento successivo (Figura 14.3b). In c) viene richiesta all'utente l'immissione del prossimo valore della sequenza e tale valore viene assegnato al campo informazione dell'elemento (Figura 14.3c). Il ciclo verrà ripetuto un'altra volta ottenendo il risultato di Figura 14.4.

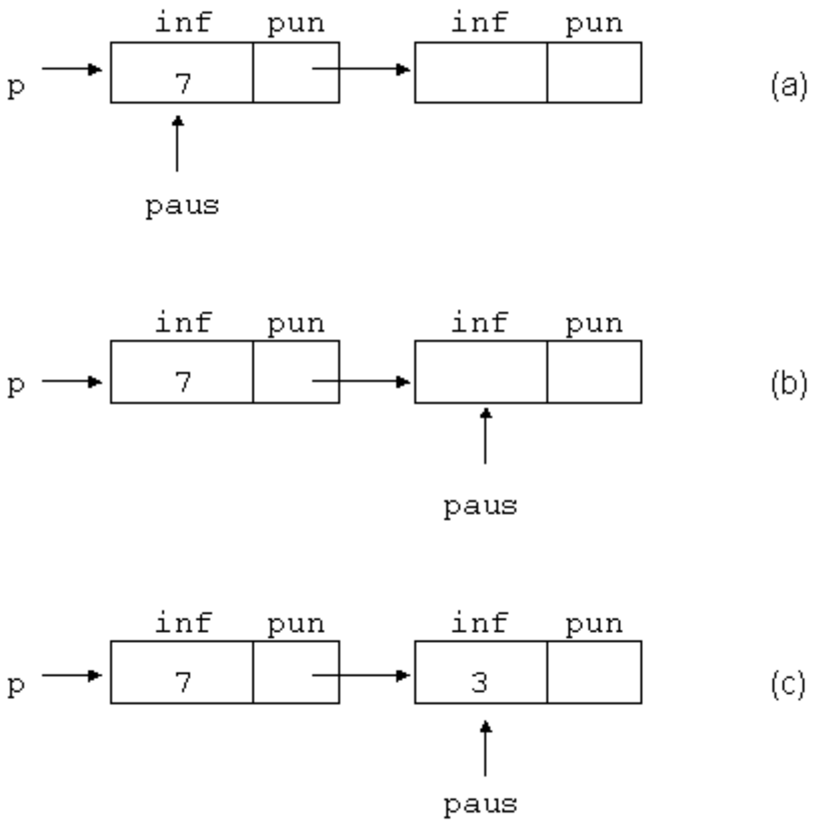


Figura 14.3 Sequenza di creazione del secondo elemento della lista

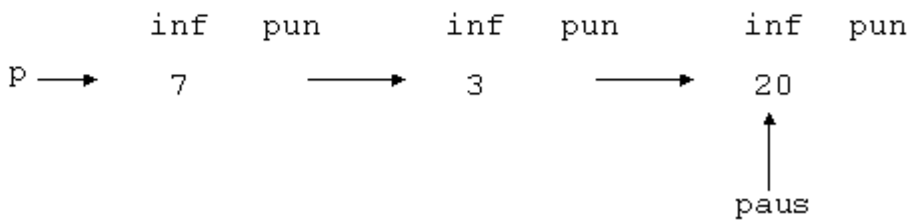


Figura 14.4 Situazione dopo l'ultimo ciclo

Infine la funzione assegna al campo puntatore dell'ultimo elemento il valore `NULL` e termina passando al chiamante il valore di `p`, cioè il puntatore alla lista:

```
return(p);
```

Il controllo passa quindi al `main`, che ora dispone della lista configurata come in Figura 14.5. Le variabili `paus` e `n` non esistono più perché sono state dichiarate locali alla procedura.

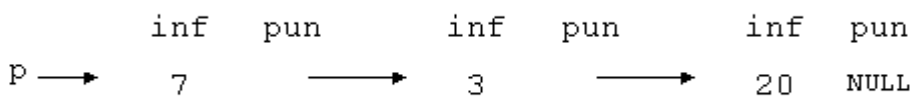


Figura 14.5 Lista completa

La procedura `visualizza_lista` effettua la scansione. È previsto un ciclo che visualizzi il campo informazione di ogni elemento a iniziare dal primo; la funzione è stata dichiarata di tipo `void` poiché non restituisce un valore di ritorno.

Il `main` passa alla procedura il puntatore iniziale alla lista:

```
visualizza_lista(punt_lista);
```

Il parametro attuale `punt_lista` corrisponde al parametro formale `p`. Per effettuare la scansione della lista utilizziamo il seguente ciclo:

```
while(p!=NULL) {
  printf("%d", p->inf);    /* Visualizza il campo
                              informazione */
  printf("---> ");
  p = p->pun;              /* Scorri di un elemento
                              in avanti */
}
```

Il ciclo di scansione è controllato dal test sopra il puntatore `p`: se `p!=NULL` continua l'iterazione. Questo controllo ci è permesso perché abbiamo avuto cura di porre il segnale di fine lista nella funzione `crea_lista`. La scansione degli elementi è consentita dall'operazione di assegnamento:

```
p = p->pun;
```

Si provi a eseguire manualmente l'algoritmo di scansione sulla lista di Figura 14.5. La procedura funziona anche nel caso particolare di lista vuota. Non c'è bisogno di un puntatore ausiliario, dato che il riferimento all'inizio della lista è nella variabile `punt_lista` del `main`, mentre `p` è una variabile locale al sottoprogramma ▬.

```
/* Accetta in ingresso una sequenza di interi e li memorizza in
   una lista. Il numero di interi che compongono la sequenza
   è richiesto all'utente. La lista creata viene visualizzata */

#include <stdio.h>
#include <malloc.h>

/* struttura degli elementi della lista */
struct elemento {
  int inf;
  struct elemento *pun;
};

struct elemento *crea_lista();
void visualizza_lista(struct elemento *);

main()
{
struct elemento *punt_lista;  /* Puntatore alla testa
                                 della lista           */
punt_lista = crea_lista();    /* Chiamata funzione per
                                 creare la lista       */
visualizza_lista(punt_lista); /* Chiamata funzione per
                                 visualizzare la lista */
}


/* Funzione per l'accettazione dei valori immessi
   e la creazione della lista. Restituisce il puntatore alla testa */
struct elemento *crea_lista()
{
struct elemento *p, *paus;
int i, n;
```

```
printf("\n Di quanti elementi è composta la sequenza?  ");
scanf("%d", &n);

if(n==0)  p = NULL;           /* lista vuota */
else
{
  /* Creazione del primo elemento */
  p = (struct elemento *)malloc(sizeof(struct elemento));
  printf("\nInserisci la 1 informazione: ");
  scanf("%d", &p->inf);
  paus = p;

  /* creazione degli elementi successivi */
  for(i=2; i<=n; i++) {
   paus->pun = (struct elemento *)malloc(sizeof(struct elemento));
   paus = paus->pun;
   printf("\nInserisci la %d informazione: ", i);
   scanf("%d", &paus->inf);
  }
  paus->pun = NULL;           /* Marca di fine lista */
}
return(p);
}


/* Funzione per la visualizzazione della lista.
   Il parametro in ingresso è il puntatore alla testa */
void visualizza_lista(struct elemento *p)
{
printf("\npunt_lista---> ");

/* Ciclo di scansione della lista */
while(p!=NULL) {
  printf("%d", p->inf);    /* Visualizza il campo informazione */
  printf("---> ");
  p = p->pun;              /* Scorri di un elemento in avanti  */
}
printf("NULL\n\n");
}
```

Listato 14.1 Creazione e visualizzazione di una lista

14.4 Determinazione del maggiore di una lista

Consideriamo il seguente problema: memorizzare una sequenza di numeri interi terminante con zero in una lista lineare, visualizzare la lista e determinare il maggiore degli elementi. Il valore zero non fa parte della lista.
Dividiamo il problema in tre sottoproblemi:

- • memorizzare la sequenza in una lista lineare;
- • visualizzare la lista;
- • determinare il maggiore della sequenza.

Le dichiarazioni necessarie sono le seguenti:

```
struct elemento {
       int inf;
       struct elemento *pun;
```

```
    };

    elemento *crea_lista2();
    void visualizza_lista(struct elemento *);
    int maggiore_lista(struct elemento *);
```

Rispetto al problema precedente si dovrà modificare la funzione `crea_lista`, poiché il numero degli elementi della sequenza non è indicato a priori dall'utente; la funzione `maggiore_lista` va sviluppata. Nel Listato 14.2 viene riportata la parte nuova di programma.

Nel resto del capitolo, le funzioni già esaminate precedentemente non verranno ridefinite ma solo dichiarate e invocate.

Evidenziamo le differenze tra `crea_lista2` e `crea_lista` vista anteriormente. Questa volta non possiamo inserire direttamente l'informazione passata dall'utente in un elemento di lista mediante l'istruzione `scanf("%d", &p->inf)`, perché non sappiamo se questa fa parte effettivamente della sequenza o se corrisponde alla marca di fine sequenza (valore zero). A questo scopo utilizziamo la variabile `x` di tipo `struct elemento` per accettare i valori in entrata.

Si richiede all'utente di inserire la prima informazione: se questa è uguale a zero, si assegna a `p` il valore `NULL` e la funzione ha termine:

```
printf("\nInserisci un'informazione (0 per fine lista): ");
    scanf("%d", &x.inf);
    if(x.inf==0)  p = NULL;
```

Nel caso in cui il valore della prima informazione in entrata sia diverso da zero si crea il primo elemento di lista, si assegna `x.inf` e si inizializza il puntatore ausiliario `paus` al valore `p` (`punt_lista`):

```
p = (struct elemento *)malloc(sizeof(struct elemento));
    p->inf = x.inf;
    paus = p;
```

Inizia poi il ciclo di ingresso delle altre informazioni. Se il valore dell'informazione è diverso da zero si procede alla creazione di un nuovo elemento di lista e all'inserimento dell'informazione nel corrispondente campo:

```
/* creazione dell'elemento successivo */
paus->pun = (struct elemento *)malloc(sizeof(struct elemento));

paus = paus->pun;                  /* Attualizzazione di paus */
paus->inf = x.inf;                 /* Inserimento dell'informazione
                                      nell'elemento */
```

La soluzione è simile a quella vista per la visualizzazione. Si tratta di scandire la lista per cercare il valore maggiore. Il `main` passa alla funzione il puntatore alla lista `punt_lista` e il valore di ritorno della funzione è il maggiore che viene visualizzato:

```
printf("\nIl maggiore è: %d", maggiore_lista(punt_lista));
```

All'interno di `maggiore_lista` dichiariamo la variabile `max` di tipo `int`, dove inseriamo e successivamente visualizziamo il maggiore fra gli elementi; `max` deve essere inizializzato a `MIN_INT`, costante definita in `limits.h`, contenente il minimo intero rappresentabile.

Si scorre la lista lineare nello stesso modo della funzione `visualizza_lista`, ma a ogni ciclo si controlla se il campo informazione dell'elemento considerato sia maggiore di `max`, nel qual caso si effettua l'assegnamento:

```
max = p->inf;
```

```
/* Accetta in ingresso una sequenza di interi e li
```

```
   memorizza in una lista. La sequenza termina quando
   viene immesso il valore zero. La lista creata viene
   visualizzata. Determina il maggiore della lista   */

#include <stdio.h>
#include <malloc.h>
#include <limits.h>

struct elemento {
  int inf;
  struct elemento *pun;
};

struct elemento *crea_lista2();
void visualizza_lista(struct elemento *);
int maggiore_lista(struct elemento *);

main()
{
struct elemento *punt_lista;  /* Puntatore alla testa
                                 della lista           */
punt_lista = crea_lista2();   /* Chiamata funzione per
                                 creare la lista       */
visualizza_lista(punt_lista); /* Chiamata funzione per
                                 visualizzare la lista */
/* Stampa il valore di ritorno della funzione maggiore_lista() */
printf("\nIl maggiore e': %d\n\n", maggiore_lista(punt_lista));
}


/* Accetta in ingresso una sequenza  di interi e li
   memorizza in una lista. Il numero di interi che compongono
   la sequenza termina con il valore zero                     */
struct elemento *crea_lista2()
{
struct elemento *p, *paus;
struct elemento x;

printf("\nInserisci un'informazione (0 per fine lista): ");
scanf("%d", &x.inf);

if(x.inf==0)  p = NULL;           /* Lista vuota */
else {
/* Creazione del primo elemento */
 p = (struct elemento *)malloc(sizeof(struct elemento));

 p->inf = x.inf;
 paus=p;

 while(x.inf!=0) {
  printf("\nInserisci un'informazione (0 per fine lista): ");
  scanf("%d", &x.inf);
  if(x.inf!=0) {
   /* creazione dell'elemento successivo */
   paus->pun = (struct elemento *)malloc(sizeof(struct elemento));

   paus = paus->pun;      /* Attualizzazione di paus */
   paus->inf = x.inf;     /* Inserimento dell'informazione
                             nell'elemento                 */
```

```
  }
  else
   paus->pun = NULL;      /* Marca di fine lista */
 }
}
return(p);
}


/* Determina il maggiore della lista.
   Il parametro in ingresso è il puntatore alla testa */
maggiore_lista(struct elemento *p)
{
int max = INT_MIN;

/* Ciclo di scansione della lista */
while(p != NULL) {
  if(p->inf > max)
    max = p->inf;
  p = p->pun;             /* Scorre di un elemento in avanti */
}
return(max);
}

/* Visualizza la lista */
void visualizza_lista(struct elemento *p)
{
printf("\npunt_lista---> ");

/* Ciclo di scansione della lista */
while(p!=NULL) {
  printf("%d", p->inf);    /* Visualizza il campo informazione */
  printf("---> ");
  p = p->pun;              /* Scorre di un elemento in avanti */
}
printf("NULL\n\n");
}
```

Listato 14.2 Programma che crea, visualizza la lista e ne determina il maggiore. L'immissione dei valori da parte dell'utente termina con zero

14.5 Somma tra liste

Consideriamo ora il problema di memorizzare due sequenze di numeri interi in due liste lineari, visualizzare le due liste e assegnare la somma delle corrispondenti informazioni delle due liste in una terza lista lineare che infine è anch'essa da visualizzare.
La lista somma avrà lunghezza uguale alla minore delle altre due: se la prima sequenza è: 2, 100, 20, 0 e la seconda sequenza è: 7, 300, 9, 527, 0, la lista somma sarà composta dai valori 9, 400, 29. Suddividiamo il problema nei seguenti sottoproblemi:

- • creare la prima lista;
- • creare la seconda lista;
- • visualizzare la prima lista;
- • visualizzare la seconda lista;
- • creare la terza lista inserendo nei suoi elementi il risultato della somma delle altre due;
- • visualizzare la terza lista.

Per creare le due liste possiamo utilizzare la funzione `crea_lista2` vista nel paragrafo precedente, chiamandola due volte nel `main`; la prima volta il puntatore restituito verrà assegnato a `punt_lista1`, la seconda a `punt_lista2`:

```
printf("\n PRIMA LISTA \n");
     punt_lista1 = crea_lista2();

     printf("\n SECONDA LISTA \n");
     punt_lista2 = crea_lista2();
```

Procederemo analogamente per visualizzare le due liste:

```
visualizza_lista(punt_lista1);
     visualizza_lista(punt_lista2);
```

Per costruire la lista risultato della somma delle prime due costruiremo la funzione `somma_liste`,
Quest'ultima riceve in ingresso i puntatori alle due liste da sommare e restituisce il puntatore alla nuova lista:

```
punt_lista3 = somma_liste(punt_lista1, punt_lista2);
```

Utilizzeremo la procedura `visualizza_lista` per visualizzarla:

```
visualizza_lista(punt_lista3);
```

Nel Listato 14.3 viene riportato il `main` e la funzione `somma_liste`.
Si tratta di scandire in parallelo le due liste, sommare il campo informazione degli elementi corrispondenti, creare un elemento della terza lista e inserivi il risultato ottenuto. Si effettueranno somme finché una delle due liste non termina. I parametri attuali `punt_lista1`, `punt_lista2` diventano i parametri formali `p1`, `p2` e il valore di ritorno `punt_lista3` è il puntatore alla lista somma delle precedenti:

```
punt_lista3 = somma_liste(punt_lista1, punt_lista2);
```

Si utilizzano due puntatori `p1` e `p2` per visitare le due liste, `p3` per creare il primo elemento della terza lista e `paus3` per creare i successivi. Il ciclo ha termine quando `paus1` o `paus2` diventa uguale a `NULL`. Il puntatore `p3` alla terza lista viene restituito al programma chiamante.

```
/* Crea due liste e le visualizza. Somma gli elementi
corrispondenti delle due liste, inserisce il risultato
in una terza lista e la visualizza                    */

#include <stdio.h>
#include <malloc.h>

struct elemento {
  int inf;
  struct elemento *pun;
};

struct elemento *crea_lista2();
void visualizza_lista(struct elemento *);
struct elemento *somma_liste(struct elemento *,struct elemento *);

main()
{
struct elemento *punt_lista1, *punt_lista2, *punt_lista3;

printf("\n PRIMA LISTA \n");
punt_lista1 = crea_lista2();
```

```
printf("\n SECONDA LISTA \n");
punt_lista2 = crea_lista2();

visualizza_lista(punt_lista1);
visualizza_lista(punt_lista2);

/* Invocazione della funzione per la somma delle liste */
punt_lista3 = somma_liste(punt_lista1, punt_lista2);

/* Visualizzazione della lista somma delle precedenti  */
visualizza_lista(punt_lista3);
}


/*  Somma gli elementi corrispondenti di due liste
e inserisce il risultato in una terza lista        */
struct elemento *somma_liste(struct elemento *p1, struct elemento *p2)
{
struct elemento *p3 = NULL, *p3aus;

if(p1!=NULL && p2!=NULL) {
                      /* Creazione primo elemento  */
 p3 = (struct elemento *)malloc(sizeof(struct elemento));
 p3->inf = p1->inf + p2->inf;             /* somma */
 p3aus = p3;              /* p3aus punta III lista */
 p1 = p1->pun;            /* Scorrimento I lista   */
 p2 = p2->pun;            /* Scorrimento II lista  */

 /* Creazione elementi successivi */
 for(; p1!=NULL && p2!=NULL;) {
  p3aus->pun = (struct elemento *)malloc(sizeof(struct elemento));
  p3aus = p3aus->pun;     /* Scorrimento III lista */
  p3aus->inf = p1->inf + p2->inf;         /* Somma */
  p1 = p1->pun;           /* Scorrimento I lista   */
  p2 = p2->pun;           /* Scorrimento II lista  */
 }
 p3aus->pun = NULL;     /* Marca di fine III lista */
}
return(p3);           /* Ritorno del puntatore alla III lista */
}

/* ATTENZIONE: devono essere aggiunte le definizioni delle
funzioni per creare - crea_lista2() - e visualizzare
– visualizza_lista() - la lista presenti nel precedente Listato 14.2
*/
```

Listato 14.3 Somma dei valori di due liste

14.6 Soluzioni ricorsive

I problemi che implicano lo scorrimento della lista lineare possono essere risolti elegantemente con funzioni ricorsive: ogni elemento contiene un puntatore a un oggetto identico a se stesso (in questo senso si parla di oggetti ricorsivi).
Alla funzione ricorsiva `vis2_lista` che visualizza la lista si passa il puntatore iniziale:

```
vis2_lista(punt_lista);
```

La funzione si domanda se la scansione della lista sia terminata, nel qual caso stampa la parte informazione seguita da una freccia che indica graficamente il puntatore al prossimo elemento e chiama ricorsivamente `vis2_lista` passandogli il puntatore `p->pun` all'elemento successivo:

```
void vis2_lista(struct elemento *p)
     {
     if(p!=NULL) {
       printf("%d", p->inf);
       printf("---> ");
       vis2_lista(p->pun);
     }
     else
       printf("NULL");
     }
```

L'ultima chiamata avviene quando la lista è terminata e la funzione stampa `NULL` per indicare questo fatto.
Analoghe considerazioni si possono fare per la funzione che determina il maggiore della lista, dato che anche in questo caso si tratta di scandirla. La differenza consiste nel fatto che la variabile `max` che contiene il maggiore deve essere inizializzata fuori dalla funzione, prima della sua chiamata:

```
maggiore = INT_MIN;
     printf("\nIl maggiore è: %d", mag2_lista(punt_lista, maggiore));
```

Ovviamente `maggiore` è una variabile intera:

```
mag2_lista(struct elemento *p, int max)
     {
     if(p!=NULL) {
       if(p->inf>max)
         max = p->inf;
       max = mag2_lista(p->pun, max);
     }
     return(max);
     }
```

14.7 Pila

La pila (*stack*) è una struttura composta da elementi omogenei ordinati in una sequenza. Le operazioni di inserimento ed eliminazione di un elemento sono effettuate a uno stesso estremo della sequenza, che viene detto *testa della pila*.
La pila è una struttura astratta, monodimensionale e dinamica, definita in base alle regole che governano le operazioni di inserimento ed eliminazione degli elementi. Il tipo di logica che vi si applica è detto LIFO (*Last In First Out*): l'ultimo elemento inserito è quello che per primo viene eliminato. Un esempio del metodo descritto è quello che viene applicato su un binario terminale, dove sono immessi e prelevati i vagoni in deposito: ogni singola carrozza viene inserita o prelevata dal binario morto per mezzo di una locomotiva che la spinge; l'ultima carrozza inserita è la prima a essere prelevata. Allo scopo di simulare la gestione del binario terminale si supponga che ogni vagone sia codificato mediante un carattere. Se arrivano al deposito, in ordine di tempo, i vagoni: D, F, C e R, la sequenza è così composta:

`D F C R <---` testa della pila

La testa della pila corrisponde all'ultimo elemento in essa inserito. Se, adesso, si viene incaricati di prelevare un vagone, la sequenza diventa:

`D F C <---` testa della pila

Si è cioè prelevato dalla sequenza l'ultimo elemento inserito, quello in testa alla pila. Se poi si viene incaricati di un ulteriore prelievo, la sequenza diventa:

`D F <---` testa della pila

L'arrivo del vagone codificato con il carattere A corrisponde invece a un inserimento in testa alla pila:

`D F A <---` testa della pila

Dunque, come detto, le inserzioni e le estrazioni avvengono sempre in testa alla pila. Un altro esempio di pila è dato dalle bambole russe, le famose *matrioske*, che vengono chiuse una sull'altra, in un ordine che va dalla più piccola alla più grande. Se consideriamo la sequenza delle bambole già racchiuse, ci rendiamo conto che possiamo effettuare solo due operazioni: inserire una bambola più grande in testa alla sequenza ed eliminare la più grande delle bambole presenti nella sequenza, appunto quella che è stata inserita per ultima.

14.8 Gestione di una pila mediante array

Passiamo ora a considerare il problema seguente: far gestire all'utente una struttura astratta di tipo pila, per mezzo delle operazioni di inserimento ed eliminazione di elementi, e visualizzare la pila stessa. La parte informazione di un elemento della pila è costituito da un valore intero. Bisogna implementare la pila per mezzo di un array. La soluzione del problema ha una notevole valenza didattica, perché mostra visivamente la logica di funzionamento di una pila.
Dovendo utilizzare un array, è necessario stimare il numero massimo di elementi che la pila può contenere. Si utilizzerà la variabile di tipo intero `punt_testa` come indice (riferimento) alla testa della pila, cioè all'elemento dove si effettuerà il prossimo inserimento (Figura 14.6). Se si suppone che sia n=5, `punt_testa` potrà assumere valori compresi da 0 a 5, dove 0 indicherà pila vuota, 5 pila piena. Se `pila` è il nome dell'array le dichiarazioni corrispondenti sono:

```
#define LUN_PILA 4
     int pila[LUN_PILA];
     int punt_testa = 0;
```

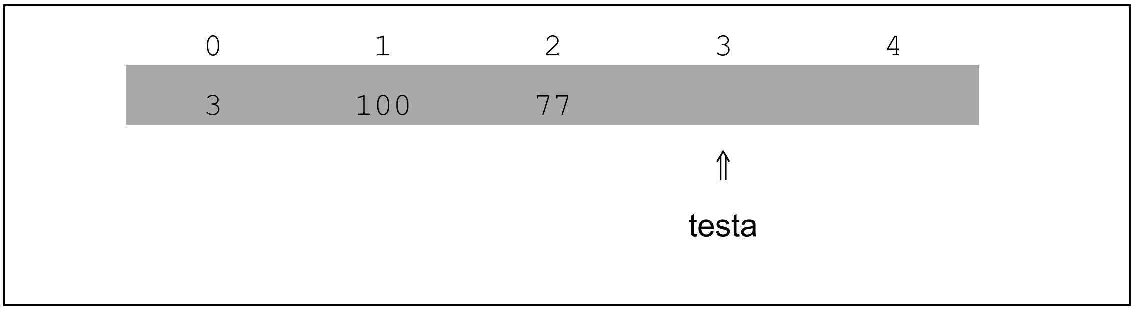


Figura 14.6 Implementazione di una pila mediante un array; il valore del puntatore alla testa è 3

Una prima suddivisione del problema porta a individuare i seguenti sottoproblemi:

- • inserzione in testa alla pila (*push*);
- • eliminazione in testa alla pila (*pop*);
- • visualizzazione della pila.

Nel problema si richiede che l'utente gestisca la pila, ovvero si possano effettuare le tre operazioni in qualsiasi sequenza, un numero qualunque di volte. Per facilitare il compito si visualizzano le opzioni in un menu con le possibili scelte:

```
        ESEMPIO UTILIZZO STRUTTURA ASTRATTA: PILA

                 1. Per inserire un elemento
                 2. Per eliminare un elemento
                 3. Per visualizzare la pila
                 0. Per finire

                  Scegliere una opzione:
```

L'utente deve inserire il numero corrispondente all'opzione desiderata (1 per inserire, 2 per eliminare ecc.). La funzione `gestione_pila` presiede al trattamento della scelta:

```
while(scelta!=0) {
        visualizza menu;
        leggi l'opzione dell'utente nella variabile scelta;
        switch(scelta) {
          case 1:
              leggi l'informazione da inserire;
              esegui l'inserimento in testa alla pila;
              break;
          case 2:
              esegui l'eliminazione in testa alla pila;
              visualizza l'informazione eliminata;
              break;
          case 3:
              visualizza la pila;
              break;
     }
```

Si noti che l'operazione di inserimento di un elemento della pila non si può effettuare se la pila è piena (Figura 14.7), cioè se il numero di elementi presenti nella pila è uguale a n, 5 nel nostro caso (`punt_testa=5`). Perciò un ulteriore sottoproblema è quello di determinare se la pila è piena.

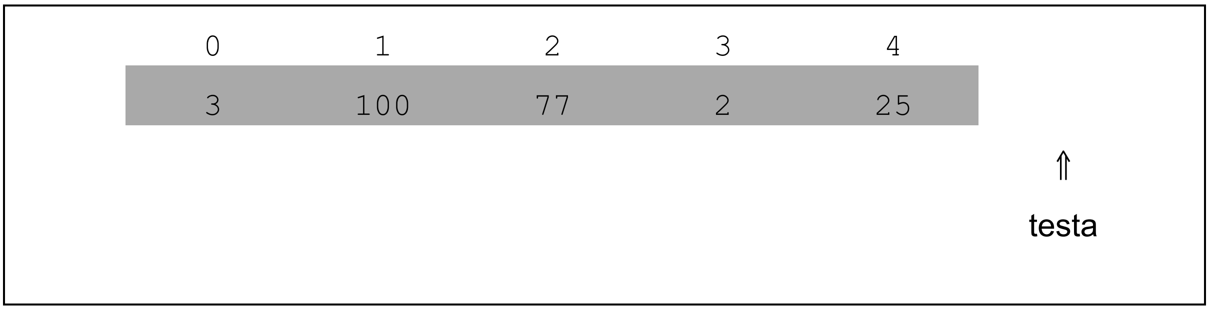


Figura 14.7 Nel caso di pila piena non è possibile effettuare un inserimento; il puntatore alla testa della pila ha valore *n*

D'altra parte l'operazione di eliminazione non si può effettuare nel caso di pila vuota. Ciò avviene quando `punt_testa` ha valore zero. Quindi un ultimo sottoproblema consiste nel determinare se la pila è vuota. Queste osservazioni permettono di dettagliare `gestione_pila`:

```
void gestione_pila(void)
     {
     int pila[LUN_PILA];
     int punt_testa = 0;
     int scelta = -1, ele;
     char pausa;

     while(scelta!=0) {
       visualizza menu
       leggi l'opzione dell'utente nella variabile scelta;
       switch(scelta) {
         case 1:
            Se (la pila è piena):
               l'inserimento è impossibile;
            Altrimenti:
               leggi l'informazione da inserire;
               Esegui l'inserimento in testa alla pila;
            break;
         case 2:
            Se (la pila è vuota):
               l'eliminazione è impossibile;
            Altrimenti:
               esegui l'eliminazione in testa alla pila;
               visualizza l'informazione eliminata;
            break;
         case 3:
            Visualizza la pila;
            break;
        }
     }
     }
```

Non rimane che risolvere i seguenti sottoproblemi:

- inserimento di un elemento in testa alla pila;
- eliminazione di un elemento in testa alla pila;
- verifica di pila piena;
- verifica di pila vuota;
- visualizzazione della pila.

Nel Listato 14.4 riportiamo il programma completo. La chiamata della funzione `inserimento`:

```
punt_testa = inserimento(pila, &punt_testa, ele);
```

prevede il passaggio dei parametri:

- • `pila`, il puntatore alla struttura dati (array) che contiene fisicamente la pila;
- • `punt_testa`, il puntatore (un valore intero) alla testa della pila;
- • `ele`, la variabile che contiene l'elemento da inserire.

Naturalmente `ele` è passato per valore. Nella procedura questi parametri prendono i nomi di `pila, p` ed `ele`:

```
inserimento(int *pila, int *p, int ele)
```

La funzione inserimento effettua le seguenti azioni:

```
pila[*p] = ele;
     ++*p;

     return(*p);
```

Inserisce il valore di `ele` in `pila[*p]`, incrementa di uno il valore del puntatore alla testa e lo restituisce al chiamante (Figura 14.8). Si ricordi che `punt_testa` è passato per indirizzo e quindi per far riferimento al suo valore si deve scrivere `*p`.

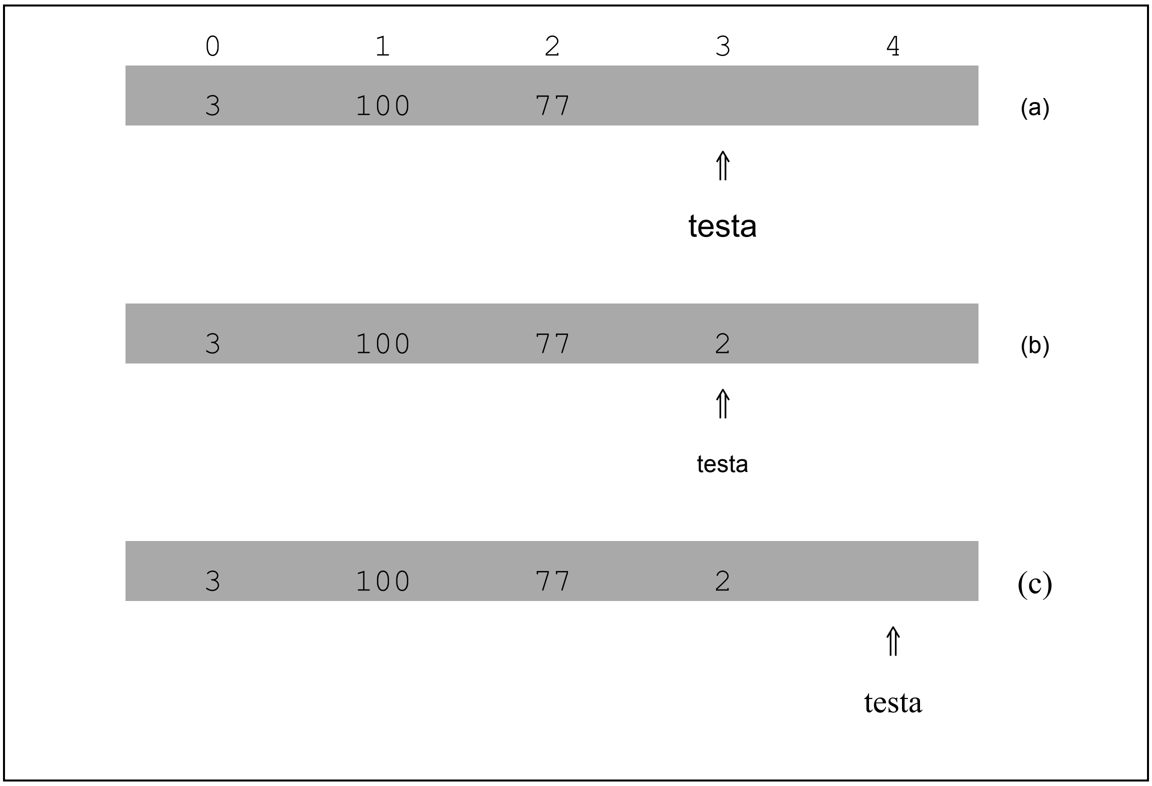


Figura 14.8 a) Stato della pila prima dell'inserimento. b) Si inserisce la nuova informazione (`ele=2`) nell'elemento dell'array indicato da `*p` (`punt_testa`). c) Si incrementa di uno il valore di `*p` (`punt_testa`)

Alla funzione di eliminazione devono essere passati gli stessi parametri della funzione `inserimento`:

```
punt_testa = eliminazione(pila, &punt_testa, &ele);
```

Questa volta il parametro `ele` viene passato per indirizzo in quanto la procedura restituisce il valore dell'elemento eliminato al `main`:

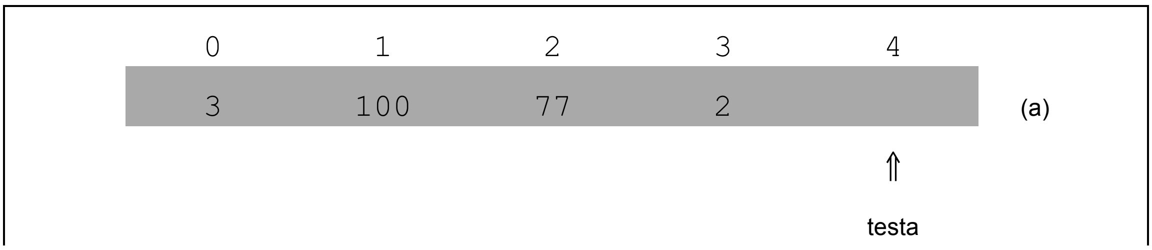
```
eliminazione(int *pila, int *p, int *ele)
```

La funzione decrementa di uno il valore di `*p`, assegna il valore dell'elemento in testa alla `pila` a `ele` e restituisce il nuovo puntatore alla testa:

```
--*p;
     *ele = pila[*p];

     return(*p);
```

Si noti che il valore dell'elemento eliminato (Figura 14.9), sebbene permanga nella struttura fisica dell'array, non fa più parte della pila, in quanto l'ultimo elemento presente nella pila è quello che precede il puntatore alla testa.



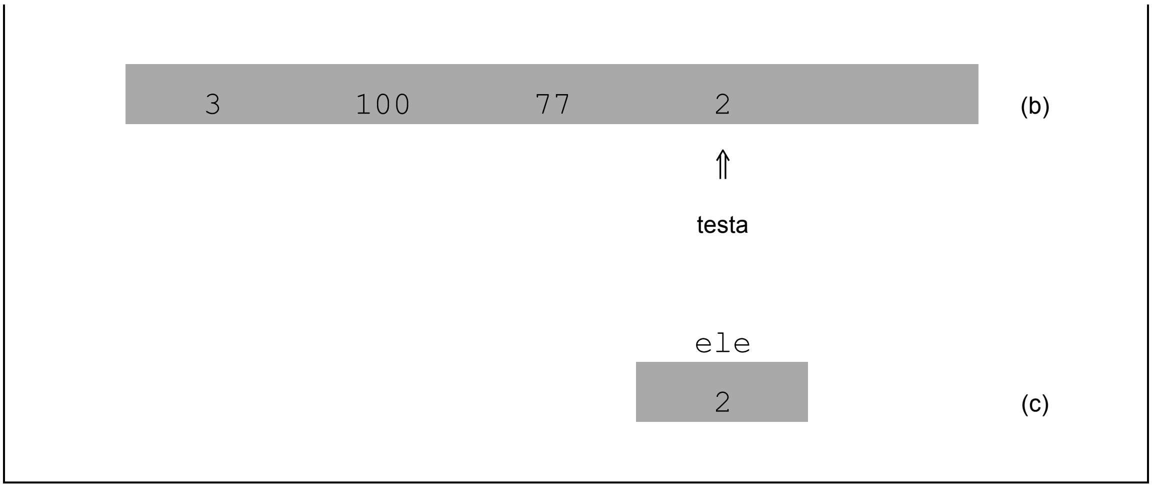


Figura 14.9 a) Stato della pila prima della eliminazione. b) Si decrementa di uno il valore di `*p` (`punt_testa`). c) L'informazione presente in testa è assegnato alla variabile `ele`

Per controllare se la pila è piena è sufficiente verificare che il puntatore alla testa sia uguale a *n*, nell'esempio *n*=5. Effettuiamo questo controllo all'interno di `gestione_pila`:

```
if(punt_testa == LUN_PILA)
        inserimento impossibile, pila piena
```

Per verificare se la pila è vuota invochiamo la funzione `pila_vuota`, passandole `punt_testa`:

```
if(pila_vuota(punt_testa) )
        eliminazione impossibile, pila vuota
     else
        eliminazione dell'elemento in testa
```

L'`if` risponde falso solamente quando l'espressione è uguale a `0`, nel qual caso è possibile procedere a una eliminazione. In effetti la funzione controlla se il puntatore alla testa è uguale a `0`, nel qual caso restituisce 1:

```
if(p==0)
       return(1);
     else
       return(0);
```

Per quel che riguarda il sottoproblema 5 – la visualizzazione della pila – si deve percorrere il vettore utilizzando un indice che va da 1 al valore del puntatore alla testa della pila, stampando di volta in volta il suo valore.

```
/* GESTIONE DI UNA PILA
   Operazioni di inserimento, eliminazione e
   visualizzazione. Utilizza un array di strutture
   per implementare la pila                       */

#include <stdio.h>
#include <malloc.h>

#define LUN_PILA 10

void gestione_pila(void);
inserimento(int *, int *, int);
eliminazione(int *, int *, int *);
pila_vuota(int);
void visualizzazione_pila(int *, int);
```

```
main()
{
gestione_pila();
}


void gestione_pila(void)
{
int pila[LUN_PILA];
int punt_testa = 0;
int scelta = -1, ele;
char pausa;

while(scelta!=0) {
  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
  printf("\t\tESEMPIO UTILIZZO STRUTTURA ASTRATTA: PILA");
  printf("\n\n\n\t\t\t 1. Per inserire un elemento");
  printf("\n\n\t\t\t 2. Per eliminare un elemento");
  printf("\n\n\t\t\t 3. Per visualizzare la pila");
  printf("\n\n\t\t\t 0. Per finire");
  printf("\n\n\n\t\t\t\t Scegliere una opzione: ");
  scanf("%d", &scelta);
  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

  switch( scelta ) {
    case 1:
       if(punt_testa >= LUN_PILA) {
          printf("Inserimento impossibile: ");
          printf("memoria disponibile terminata");
          printf("\n\n Qualsiasi tasto per continuare...");
          scanf("%c%c", &pausa, &pausa);
       }
       else {
          printf("Inserire un elemento: ");
          scanf("%d", &ele);
          punt_testa = inserimento(pila, &punt_testa, ele);
       }
       break;
    case 2:
       if(pila_vuota(punt_testa)) {
          printf("Eliminazione impossibile: pila vuota");
          printf("\n\n Qualsiasi tasto per continuare...");
          scanf("%c%c", &pausa, &pausa);
       }
       else {
          punt_testa = eliminazione(pila, &punt_testa, &ele);
          printf("Eliminato: %d", ele );
          printf("\n\n Qualsiasi tasto per continuare...");
          scanf("%c%c", &pausa, &pausa);
       }
       break;
    case 3:
       visualizzazione_pila(pila, punt_testa);
       printf("\n\n Qualsiasi tasto per continuare...");
       scanf("%c%c", &pausa, &pausa);
       break;
  }
}
```

```
}


void visualizzazione_pila(int *pila, int p)
{
printf("\n<------ Testa della pila ");
while(p>=1)
  printf("\n%d", pila[--p]);
}


inserimento(int *pila, int *p, int ele)
{
pila[*p] = ele;
++*p;
return(*p);
}


eliminazione(int *pila, int *p, int *ele)
{
--*p;
*ele = pila[*p];
return(*p);
}


int pila_vuota(int p)
{
if(p==0)
  return(1);
else
  return(0);
}
```

Listato 14.4 Gestione di una pila implementata mediante un array

14.9 Gestione di una pila mediante lista lineare

Il problema di questo paragrafo è: gestire una struttura astratta di tipo pila per mezzo delle operazioni di inserimento ed eliminazione degli elementi e visualizzazione della pila. L'informazione presente in un elemento della pila è di tipo intero. Bisogna implementare la pila per mezzo di una struttura a lista lineare.

L'analisi del problema, la scomposizione in sottoproblemi e la struttura del programma sono analoghe a quelle esaminate nel paragrafo precedente. Perciò la trattazione che segue si limita a individuare le differenze rispetto a quelle, soffermandosi sull'implementazione con la lista lineare. In questo caso non si ha bisogno di dimensionare la struttura, si utilizzerà tanto spazio in memoria quanto necessario.

La verifica di pila piena deve essere modificata perché l'allocazione dinamica della memoria non richiede una predefinizione di lunghezza massima; ciò nonostante la memoria disponibile potrebbe a un certo punto finire, caso che deve essere gestito dal programmatore. A questo proposito si veda la funzione `inserimento`. La verifica di pila vuota è banale: si tratta di verificare se il puntatore alla lista è uguale a `NULL`. La visualizzazione della pila corrisponde a una scansione della lista che abbiamo già più volte incontrato. Nel Listato 14.5 riportiamo il programma completo.

La funzione `gestione_pila` passa alla funzione `inserimento` il puntatore alla testa della lista e l'informazione da inserire:

```
punt_testa = inserimento(punt_testa, ele);
```

I parametri formali corrispondenti si chiamano `p` ed `ele`:

```
struct elemento *inserimento(struct elemento *p, int ele)
```

Deve essere creato un nuovo elemento e questo deve venire inserito in testa alla lista lineare. Per far ciò si ha bisogno di un puntatore ausiliario `paus`, per mezzo del quale viene creato il nuovo elemento (Figura 14.10a).

```
paus = (struct elemento *)malloc(sizeof(struct elemento));

     if(paus==NULL) return(NULL);
```

Se la memoria dinamica a disposizione è terminata il sistema ritorna `NULL`, la funzione si chiude e restituisce questo valore al programma chiamante, che comunica all'utente l'impossibilità di effettuare l'inserimento. Altrimenti si deve connettere l'elemento al puntatore alla testa della pila (Figura 14.10b):

```
paus->pun = p;
```

Il puntatore alla testa della pila deve riferirsi al nuovo elemento creato (Figura 14.10c):

```
p = paus;
```

Finalmente inseriamo nel nuovo elemento il valore dell'informazione passata dall'utente (Figura 14.10d):

```
p->inf = ele;
```

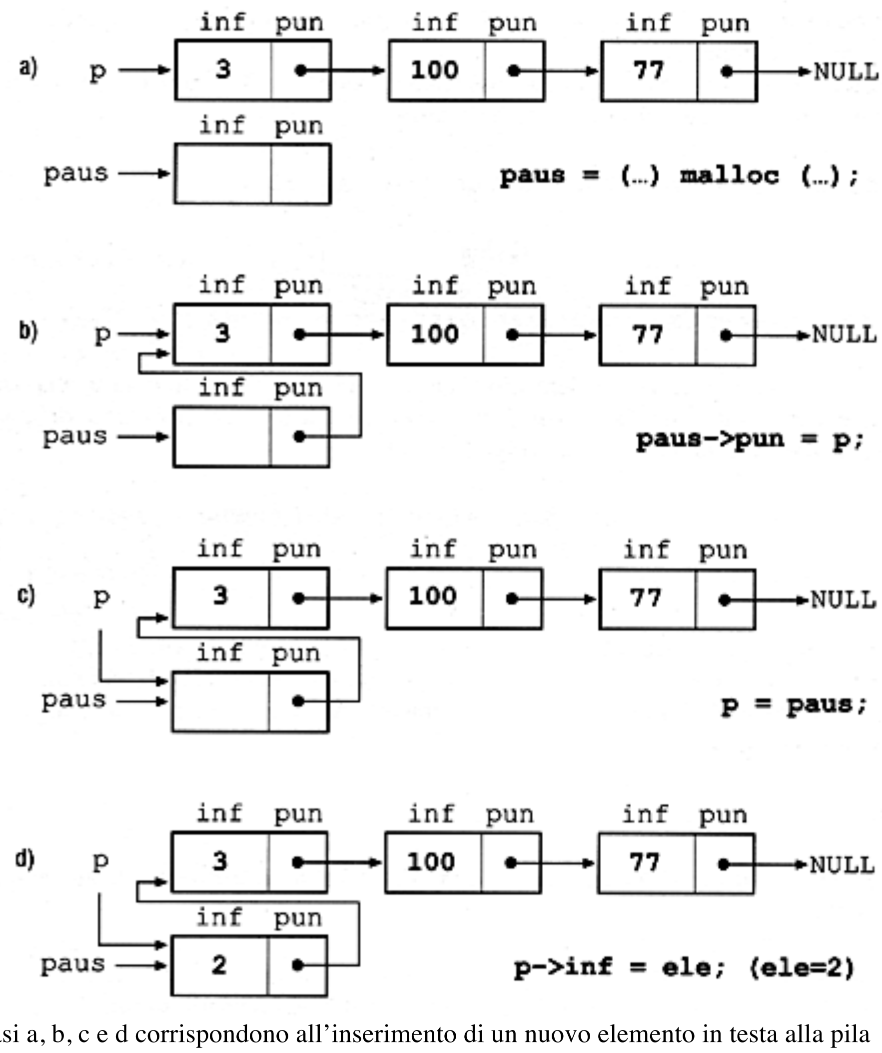


Figura 14.10 Le fasi a, b, c e d corrispondono all'inserimento di un nuovo elemento in testa alla pila

I parametri passati da `gestione_pila` a `eliminazione` sono gli stessi di `inserimento`, anche se in questo caso `ele` viene passata per indirizzo, per far in modo che la funzione restituisca l'informazione eliminata. L'eliminazione avviene semplicemente assegnando al puntatore alla lista il valore del puntatore del primo elemento (Figura 14.11a):

```
/* GESTIONE DI UNA PILA
   Operazioni di inserimento, eliminazione e
   visualizzazione. Utilizza una lista lineare
   per implementare la pila                    */

#include <stdio.h>
#include <malloc.h>

struct elemento {
  int inf;
  struct elemento *pun;
};

void gestione_pila(void);
struct elemento *inserimento(struct elemento *, int ele);
struct elemento *eliminazione(struct elemento *, int *);
int pila_vuota(struct elemento *);
void visualizzazione_pila(struct elemento *);

main()
{
  gestione_pila();
}


void gestione_pila(void)
{
struct elemento *punt_testa = NULL;
int scelta = -1, ele;
char pausa;

while(scelta!=0) {
  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
  printf("\t\tESEMPIO UTILIZZO STRUTTURA ASTRATTA: PILA");
  printf("\n\n\n\t\t\t 1. Per inserire un elemento");
  printf("\n\n\t\t\t 2. Per eliminare un elemento");
  printf("\n\n\t\t\t 3. Per visualizzare la pila");
  printf("\n\n\t\t\t 0. Per finire");
  printf("\n\n\n\t\t\t\t Scegliere una opzione: ");
  scanf("%d", &scelta);
  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

  switch(scelta) {
    case 1:
       printf("Inserire un elemento: ");
       scanf("%d", &ele);
       punt_testa = inserimento(punt_testa, ele);
       if(punt_testa==NULL) {
         printf("Inserimento impossibile: ");
         printf("memoria disponibile terminata");
         printf("\n\n Qualsiasi tasto per continuare...");
         scanf("%c%c", &pausa, &pausa);
       }
       break;
```

```
    case 2:
       if(pila_vuota(punt_testa)) {
         printf("Eliminazione impossibile: pila vuota");
         printf("\n\n Qualsiasi tasto per continuare...");
         scanf("%c%c", &pausa, &pausa);
       }
       else {
         punt_testa = eliminazione(punt_testa, &ele);
         printf("Eliminato: %d", ele );
         printf("\n\n Qualsiasi tasto per continuare...");
         scanf("%c%c", &pausa, &pausa);
       }
       break;
    case 3:
       visualizzazione_pila(punt_testa);
       printf("\n\n Qualsiasi tasto per continuare...");
       scanf("%c%c", &pausa, &pausa);
       break;
  }
}
}


void visualizzazione_pila(struct elemento *p)
{
struct elemento *paus = p;

printf("\n<------ Testa della pila ");
while(paus!=NULL) {
  printf("\n%d", paus->inf);
  paus = paus->pun;
}
}


struct elemento *inserimento(struct elemento *p, int ele)
{
struct elemento *paus;

paus = (struct elemento *)malloc(sizeof(struct elemento));

if(paus==NULL) return(NULL);

paus->pun = p;
p = paus;
p->inf = ele;
return(p);
}


struct elemento *eliminazione(struct elemento *p, int *ele)
{
struct elemento *paus;

*ele = p->inf;
paus = p;
p = p->pun;
free(paus);
return( p );
```

```
}


int pila_vuota(struct elemento *p)
{
if(p==NULL)
  return(1);
else
  return(0);
}
```

Listato 14.5 Gestione di una pila implementata mediante una lista

14.10 Coda

La coda (*queue*) è una struttura composta da una sequenza di elementi omogenei ordinati. L'operazione di inserimento di un elemento avviene dopo l'ultimo elemento inserito, quella di eliminazione è effettuata sul primo elemento della sequenza.
La coda è una struttura astratta, monodimensionale e dinamica. Il tipo di logica che si applica alle code è detto FIFO (*First In First Out*): il primo elemento inserito è quello che per primo viene estratto. Un esempio dell'applicazione del metodo descritto è dato da una fila di persone in attesa a uno sportello, in cui chi arriva primo "dovrebbe" essere servito per primo e chi arriva ultimo "dovrebbe" mettersi in coda. Usando una rappresentazione simile a quella adoperata per la pila, abbiamo:

coda `-->  D  F  C  R  -->` testa

Il prossimo elemento da servire è R, che sta in testa:

coda `-->  D  F  C  -->` testa

Il successivo è C:

coda `-->  D  F  -->` testa

Se poi arriva J, deve mettersi in coda:

coda `--->  J  D  F  --->` testa

Le implementazioni di una coda mediante un vettore e una lista sono relativamente simili a quelle viste per la pila, salvo per alcune differenze legate alla necessità di permettere, nelle code, l'accesso diretto a entrambe le estremità della sequenza.

14.11 Gestione di una coda mediante array

In prima approssimazione la gestione di una struttura astratta di tipo coda per mezzo di un vettore, con le relative operazioni di inserimento, eliminazione e visualizzazione degli elementi, ricorda la gestione di una pila; anche in questo caso dobbiamo dare una stima del massimo numero di elementi che la coda può contenere.

Una prima soluzione del problema si può in effetti ottenere con semplici modifiche a partire dal programma del Listato 14.4 con cui abbiamo implementato una pila. Di questo restano infatti invariate sia la struttura sia le variabili, i tipi di dato e tutte le funzioni che vi compaiono, con la sola eccezione delle funzioni di visualizzazione e di eliminazione. Per quanto non indispensabile, è inoltre consigliabile cambiare i nomi dei vari identificatori, sostituendo tutte le occorrenze della

parola *pila* con *coda*. In questo modo garantiremo al programma l'auspicabile, immediata leggibilità del ruolo giocato da variabili, costanti e funzioni.

Al lettore il compito di realizzare tale gestione della coda, noi passiamo invece a un'osservazione. Poiché l'inserimento e l'eliminazione (Figura 14.12) vengono effettuate agli estremi opposti della sequenza, la zona occupata dall'array tende a muoversi all'interno della struttura, allontanandosi dalla base.

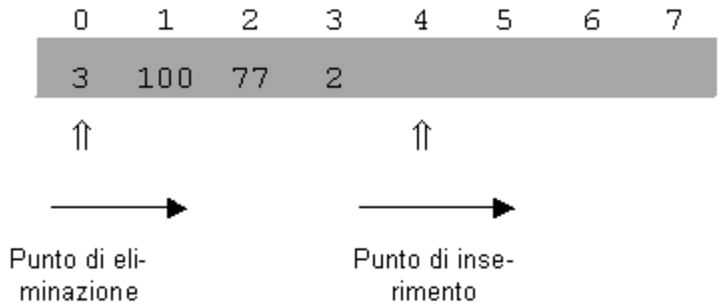


Figura 14.12 Successivi inserimenti ed eliminazioni nella coda tendono a muovere la zona occupata verso l'alto

Diventa quindi molto difficile stabilire la grandezza dell'array ed è probabile avere uno spreco di memoria. Per esemplificare quanto affermato, chiamiamo `puntCoda` il puntatore all'ultimo elemento inserito e `puntTesta` il puntatore all'elemento che deve essere estratto per primo. Dopo otto inserzioni successive avremo la coda effettivamente piena (Figura 14.13); se avvengono due eliminazioni la nuova situazione sarà quella di Figura 14.14.

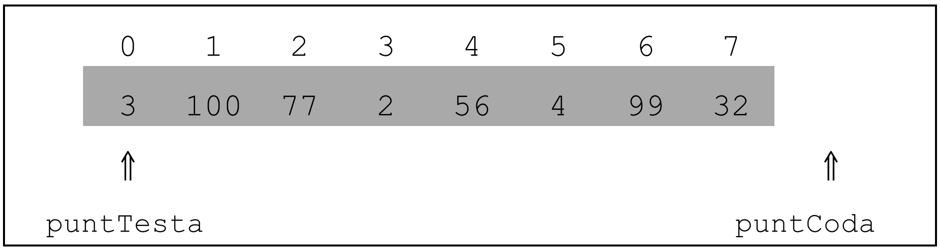


Figura 14.13 Caso di coda piena

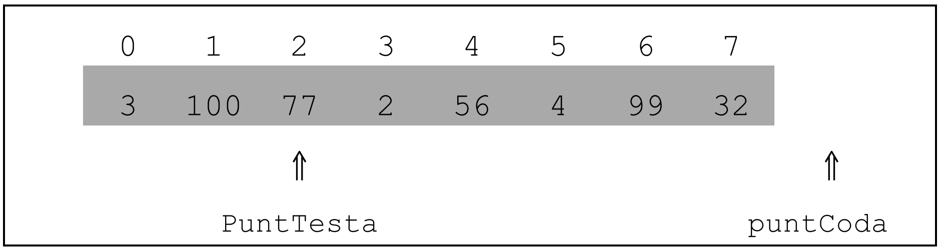


Figura 14.14 La coda ha due posizioni libere

Esistono due elementi liberi nell'array ma non sembra possibile effettuare nessun inserimento. Il fatto che in figura appaiano ancora 3 e 100 non deve indurre in errore: la sequenza significativa va dal puntatore alla testa al puntatore alla coda. Una soluzione del problema è quella di gestire le operazioni che si effettuano sull'array in modo circolare (Figura 14.15).

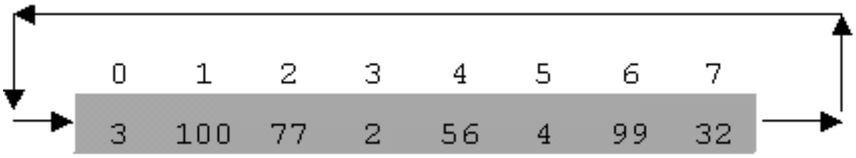


Figura 14.15 Array gestito in modo circolare

Se ritorniamo alla situazione vista in precedenza (Figura 14.14), l'inserimento di un nuovo elemento (`63`) verrebbe effettuato nella prima posizione dell'array. Per determinare l'indice dell'elemento dell'array dove effettuare l'inserimento si usa

```
puntCoda = (puntCoda % n)+1;
```

Nel caso dell'esempio `n=8` e il vecchio valore di `puntCoda` è uguale a `8`, per cui il nuovo valore di `puntCoda` è `0` (Figura 14.16).

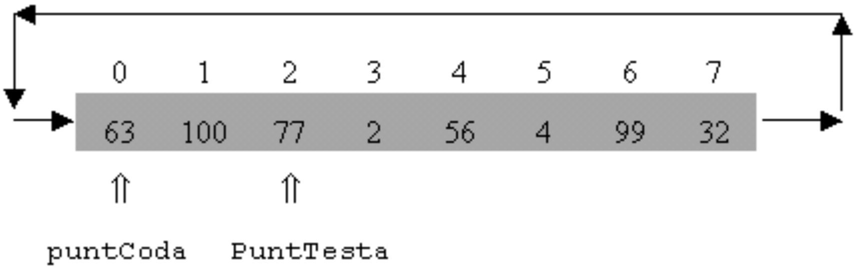


Figura 14.16 Inserimento dell'informazione (`63`) in testa alla coda

La condizione di coda piena equivale a `puntTesta` uguale a `puntCoda` più uno, considerando l'array in modo circolare, dove la posizione *n*-esima precede la prima posizione dell'array:

if(puntTesta==((puntCoda % n)+1) coda piena...

14.12 Gestione di una coda mediante liste

Consideriamo una lista che contiene gli elementi della coda in sequenza. Il puntatore `puntTesta` si riferisce alla posizione del primo elemento che verrà estratto (Figura 14.17).

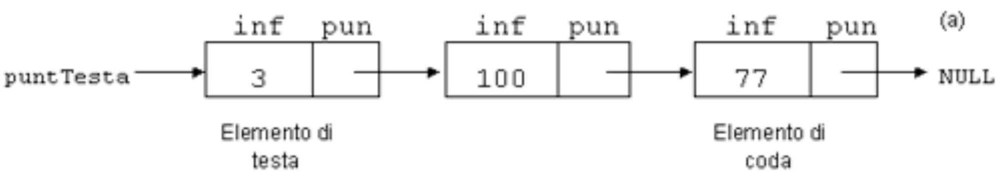


Figura 14.17 Coda implementata mediante una lista

Dunque nel caso dell'operazione di eliminazione si deve assegnare a `puntTesta` il valore del campo puntatore del primo elemento:

```
puntTesta = puntTesta->pun;
```

In questo modo viene scavalcato e dunque eliminato il primo elemento. Naturalmente si dovrebbe prevedere il rilascio della memoria corrispondente. Nel caso di inserimento si deve scorrere (scandire), mediante un puntatore ausiliario, tutta la lista, dopo di che si può effettuare l'inserimento in coda:

```
paus = puntTesta;
if(paus!=NULL) {          /* se non è vuota si visita la lista */
  while(paus->pun!=NULL) do
   paus = paus->pun;

   /* creazione del nuovo elemento      */
   paus->pun = (struct elemento *)malloc(sizeof(struct elemento));
   paus = paus->pun;      /* paus ora punta al nuovo elemento  */
   paus->inf = ele;       /* si inserisce l'informazione       */
   paus->pun = NULL;      /* si immette la marca di fine lista */
}
else {
  /* creazione primo elemento */
  puntTesta = (struct elemento *)malloc(sizeof(struct elemento));
  puntTesta->inf = ele;         /* inserisce l'informazione */
  puntTesta->pun = NULL;       /* marca di fine lista      */
}
```

Questo implica che per effettuare un inserimento si deve scandire la lista, il che è arduo se la coda contiene centinaia o migliaia di elementi. Per rendere più veloce l'operazione di inserimento si utilizza un ulteriore puntatore `punt_coda` che fa riferimento alla coda della sequenza (Figura 14.18).

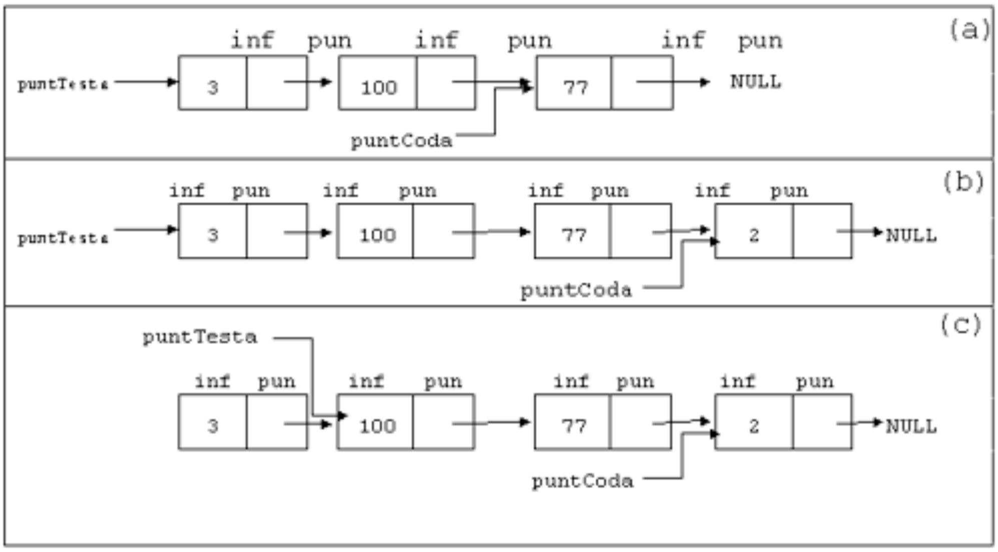


Figura 14.18 a) Implementazione di una coda mediante una lista con due puntatori. b) Inserimento di un elemento nella coda. c) Eliminazione di un elemento dalla coda

In questo caso l'operazione di inserimento è quella di Figura 14.18b; non è necessario scandire la lista poiché abbiamo attivato un puntatore alla coda:

```
/* creazione del nuovo elemento      */
puntCoda->pun = (struct elemento *)malloc(sizeof(struct elemento));
puntCoda = puntCoda->pun;  /* punt_coda al nuovo elemento */
puntCoda->inf = ele;       /* si inserisce l'informazione */
puntCoda->pun = NULL;      /* si immette la marca di fine lista */
```

Invece l'eliminazione è identica: è quella dell'implementazione precedente, che prevedeva un solo puntatore (Figura 14.18c).
Un altro metodo per gestire una coda prevede l'utilizzazione di una lista circolare nella quale l'elemento in coda punta a quello in testa (Figura 14.19a).

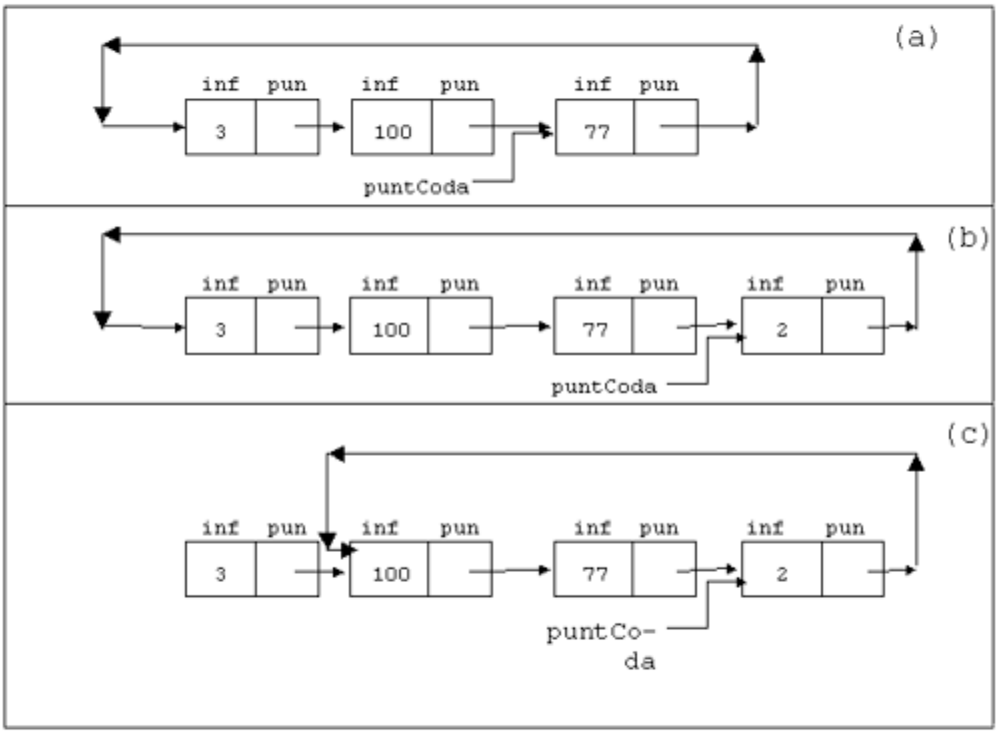


Figura 14.19 a) Implementazione di una coda mediante una lista circolare. b) Inserimento di un elemento nella coda. c) Eliminazione di un elemento dalla coda

In questo caso è necessario mantenere il solo puntatore alla coda `punt_coda`, per mezzo del quale possono essere effettuate le operazioni di inserimento (Figura 14.19b) ed eliminazione (Figura 14.19c).

```
/* INSERIMENTO creazione del nuovo elemento */
paus = (struct elemento *)malloc(sizeof(struct elemento));
paus->inf = ele;            /* inserimento dell'informazione */
paus->pun = puntCoda->pun;  /* nuovo elemento alla testa */
puntCoda->pun = paus;       /* punt_coda al nuovo elemento */
puntCoda = puntCoda->pun;   /* attualizzazione puntCoda */

/* ELIMINAZIONE */
paus = puntCoda->pun;       /* salvataggio puntatore alla testa */
puntCoda->pun = puntCoda->pun->pun;  /* eliminazione */

  free(paus);                          /* rilascio memoria */
```

14.13 Gestione di una sequenza ordinata

In questo paragrafo considereremo il problema: inserire, eliminare valori interi da una lista lineare mantenendo la lista ordinata in modo crescente; visualizzare quindi la lista. Non si conosce a priori in quale sequenza e quante volte l'utente sceglierà di effettuare le operazioni sopra indicate. Il menu è il seguente.

```
  GESTIONE DI UNA LISTA DI VALORI ORDINATI
          MEDIANTE UNA STRUTTURA A LISTA

          1. Per inserire un elemento
```

```
2. Per eliminare un elemento
3. Per visualizzare la lista
0. Per finire

  Scegliere una opzione:
```

Nel Listato 14.6 viene presentato il programma completo. In Figura 14.20 abbiamo invece l'inserimento e in Figura 14.21 l'eliminazione. Il lettore provi a disegnare una lista con valori ordinati e a eseguire manualmente passo passo le istruzioni delle funzioni.

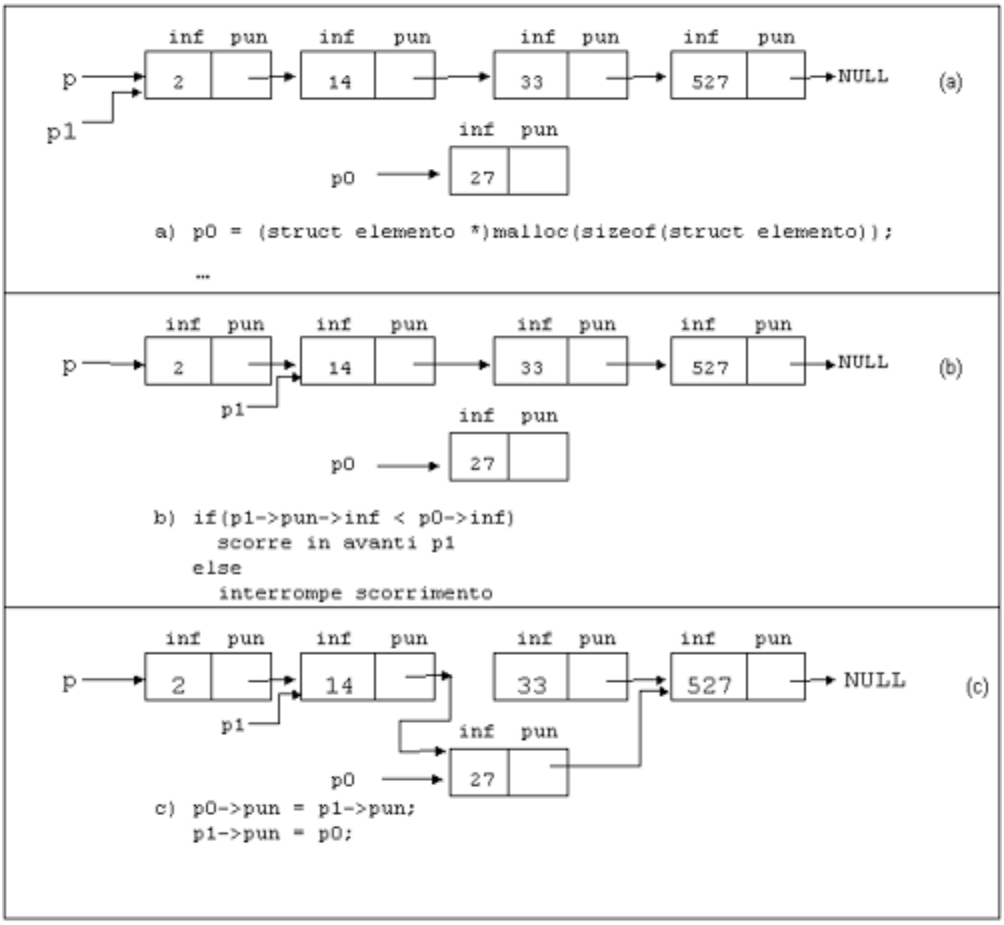


Figura 14.20 Inserimento di un elemento in una lista ordinata; nel caso preso in esame l'inserimento avviene in una posizione intermedia della lista

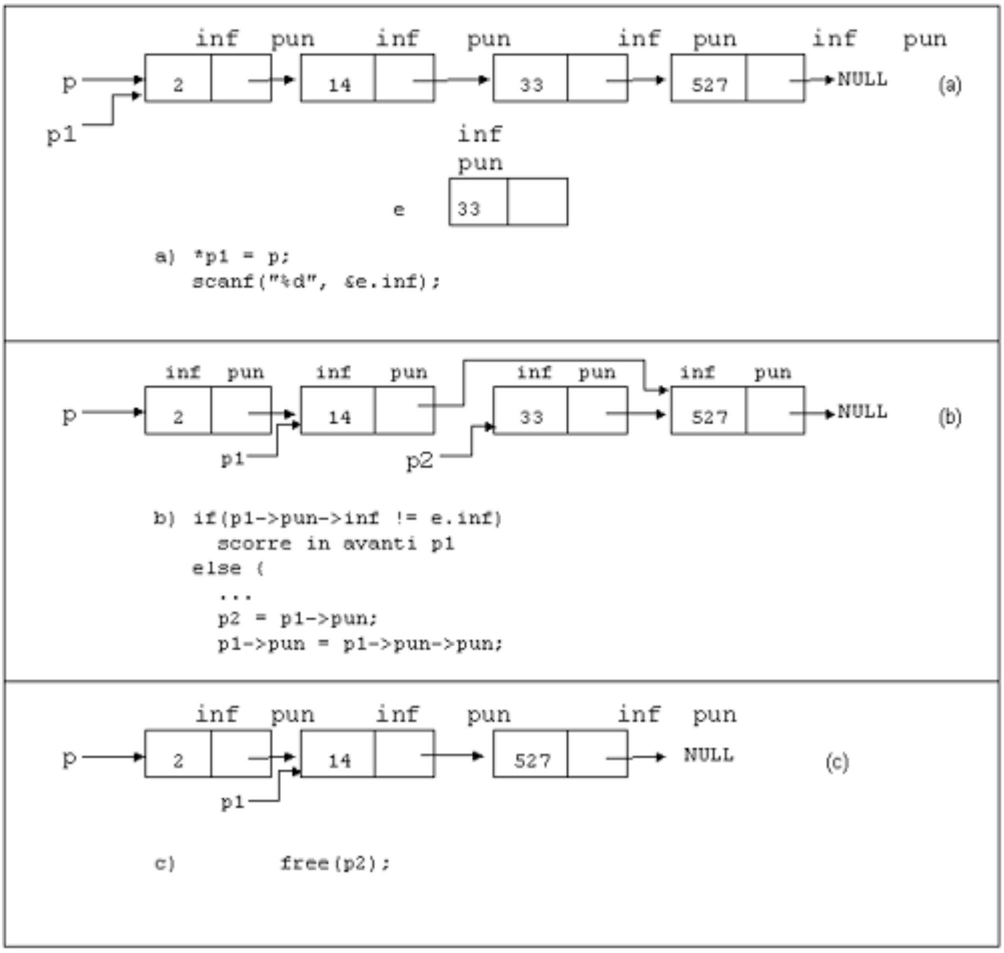


Figura 14.21 Eliminazione di un elemento da una lista ordinata; nel caso in esame l'eliminazione avviene in una posizione intermedia della lista

```
/* GESTIONE DI LISTA ORDINATA
   Operazioni di inserimento, eliminazione e visualizzazione
   Utilizza una lista lineare per implementare la pila        */

#include <stdio.h>
#include<malloc.h>

struct elemento {
  int inf;
  struct elemento *pun;
};

void gestione_lista(void);
struct elemento *inserimento(struct elemento *);
struct elemento *eliminazione(struct elemento *);
void visualizzazione(struct elemento *);

main()
{
  gestione_lista();
}


void gestione_lista(void)
{
```

```
struct elemento *punt_lista = NULL;
int scelta = -1;
char pausa;

while(scelta!=0) {
  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
  printf("\t\tGESTIONE DI UNA SEQUENZA DI VALORI ORDINATI\n");
  printf("\t\t     MEDIANTE UNA STRUTTURA A LISTA");
  printf("\n\n\n\t\t\t 1. Per inserire un elemento");
  printf("\n\n\t\t\t 2. Per eliminare un elemento");
  printf("\n\n\t\t\t 3. Per visualizzare la lista");
  printf("\n\n\t\t\t 0. Per finire");
  printf("\n\n\n\t\t\t\t Scegliere una opzione: ");
  scanf("%d", &scelta);
  printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");

  switch(scelta) {
     case 1:
        punt_lista = inserimento(punt_lista);
        break;
     case 2:
        punt_lista = eliminazione(punt_lista);
        break;
     case 3:
        visualizzazione(punt_lista);
        printf("\n\nQualsiasi tasto per continuare...");
        scanf("%c%c", &pausa, &pausa);
        break;
  }
}
}


/* Visualizzazione della lista */
void visualizzazione(struct elemento *p)
{
struct elemento *paus = p;

printf("\npunt_lista---> ");
while(paus!=NULL) {
  printf("%d---> ", paus->inf);
  paus = paus->pun;
}
printf("NULL");
}


/* Inserimento del valore passato dall'utente nella lista
   mantenendo l'ordinamento                               */
struct elemento *inserimento(struct elemento *p)
{
struct elemento *p0, *p1;
int posizione;

/* Creazione elemento */
p0 = (struct elemento *)malloc(sizeof(struct elemento));

printf("\nInserisci l'informazione (un numero intero): ");
scanf("%d", &p0->inf);
```

```
if(p==NULL) {            /* Se la lista è vuota, l'elemento      */
  p = p0;                /* diventa il primo e unico della lista */
  p->pun = NULL;
}
else {
  if(p->inf > p0->inf) {    /* Se il valore dell'elemento è  */
     p0->pun = p;           /* inferiore al primo l'elemento */
     p = p0;                /* diventa il primo della lista  */
  }
  else {           /* Ricerca della posizione di inserimento */
     p1 = p;
     posizione = 0;
     while(p1->pun!=NULL && posizione!=1) {
       if(p1->pun->inf < p0->inf)
         p1 = p1->pun;          /* Scorre in avanti p1       */
       else
         posizione = 1;         /* Interrompe lo scorrimento */
     }
     p0->pun = p1->pun; /* Connessione all'elemento successivo */
     p1->pun = p0;     /* Connessione dall'elemento precedente */
  }
}
return(p);    /* Ritorno del puntatore all'inizio della lista */
}


/* Eliminazione dell'elemento richiesto dalla lista ordinata */
struct elemento *eliminazione(struct elemento *p)
{
struct elemento *p1 = p, *p2;
struct elemento e;
int posizione = 0;
char pausa;

printf("\nInserisci l'informazione da eliminare: ");
scanf("%d", &e.inf);

if(p1!=NULL) {              /* Se la lista è vuota fine */
  if(p1->inf == e.inf) {    /* Se è il primo da eliminare */
    p2 = p1;
    p = p->pun;       /* si modifica il puntatore alla testa  */
    free(p2);
    return(p);
  }
  else {                /* Ricerca dell'elemento da eliminare */
    while(p1->pun!=NULL && posizione!=1) {
       if(p1->pun->inf!=e.inf)
          p1 = p1->pun;          /* Scorre in avanti p1 */
       else {
          posizione = 1;         /* Interrompe lo scorrimento */
          p2 = p1->pun;
          p1->pun = p1->pun->pun;    /* Eliminazione elemento */
          free(p2);                  /* Libera la memoria */
         return( p );
       }
    }
  }
}
```

```
if(!posizione) {
  printf("\nElemento non incontrato nella lista ");
  scanf("%c%c", &pausa, &pausa);
}
return(p);
}
```

Listato 14.6 Gestione di una lista ordinata

14.14 Esercizi

* 1. Scrivere un programma che accetti in ingresso una sequenza di valori interi terminante con zero e la memorizzi in una lista lineare. Successivamente il programma deve determinare il numero di pari e di dispari presenti nella lista. Risolvere il problema anche con funzioni ricorsive.

* 2. Scrivere un programma che accetti in ingresso una sequenza di valori interi terminante con zero e la memorizzi in una lista lineare. Successivamente il programma deve eliminare dalla lista i numeri pari.

3. Scrivere un programma che accetti in ingresso una sequenza di valori interi terminante con zero e la memorizzi in una lista lineare. Successivamente eliminare dalla lista creata quelli che non sono divisori di n, dove n è un numero intero passato in ingresso dall'utente.

* 4. Data in ingresso una sequenza di valori interi terminante con zero, costruire due liste lineari, una contenente i valori positivi e una i valori negativi. Visualizzare le liste costruite.

5. Definire una funzione che calcoli il numero totale degli elementi che compongono una lista. Scrivere un'altra funzione che stampi l'n-esimo elemento di una lista se questo esiste, altrimenti visualizzi il messaggio: "La lista non contiene un numero sufficiente di elementi".

6. Scrivere una funzione che cancelli da una lista tutte le occorrenze di un particolare elemento se questo è presente nella lista, altrimenti visualizzi il messaggio: "Elemento non presente nella lista".

7. Realizzare una funzione che ordini una lista in modo crescente. Scrivere un'altra funzione che inserisca in una lista ordinata al posto opportuno un nuovo elemento richiesto all'utente.

8. Scrivere una funzione che visualizzi in ordine inverso una lista.

9. Scrivere una funzione che inverta l'ordine di una lista.

10. Scrivere una funzione che, a partire da due liste, ne costruisca una terza ottenuta alternando gli elementi delle altre due.

11. Scrivere la dichiarazione di una lista bidirezionale, ovvero di una lista in cui ogni elemento, oltre al campo informazione e a quello puntatore all'elemento successivo, ha anche un campo puntatore all'elemento precedente. Definire quindi una struttura composta da elementi di questo tipo cui si può accedere grazie a due puntatori: uno al primo elemento e uno all'ultimo.

12. Scrivere un programma completo per la gestione tramite vettore di una coda. In particolare, il programma dovrà gestire le operazioni di inserimento, cancellazione e visualizzazione della coda.

13. Realizzare la gestione di una coda che comprenda le operazioni di inserimento, cancellazione e visualizzazione, implementando la struttura dati mediante una array circolare.

14. Ripetere l'esercizio precedente, implementando la struttura dati mediante una lista circolare.

15. Aggiungere alle soluzioni dei due esercizi precedenti le funzioni che permettono di verificare se un certo valore passato in ingresso dall'utente è presente nella coda.

15.1 Alberi binari

Le strutture dati che abbiamo fin qui esaminato sono lineari: ogni elemento della sequenza ha un successore e un predecessore, fatti salvi il primo e l'ultimo. Possiamo però pensare a qualcosa di più generale, in cui la relazione tra gli elementi della struttura non sia lineare.
Nel seguito utilizzeremo il termine *nodo* con il significato di elemento e il termine *arco* per indicare una connessione tra due nodi; con *etichetta* faremo invece riferimento al valore rappresentativo di ogni nodo.
Una delle strutture dati più note è l'*albero binario*, definito come un insieme B di nodi con le seguenti proprietà:

- B è vuoto oppure un nodo di B è scelto come radice;
- i rimanenti nodi possono essere ripartiti in due insiemi disgiunti B_1 e B_2, essi stessi definiti come alberi binari.

B_1 e B_2 sono detti *sottoalberi* della radice. È importante sottolineare che il sottoalbero sinistro B_1 è distinto dal sottoalbero destro B_2 e questa distinzione permane anche se uno dei due è vuoto. La ricorsività della definizione stessa è evidente.

Un esempio di albero binario è presentato in Figura 15.1. Il nodo designato come radice ha etichetta 104, da esso si diparte il sottoalbero sinistro con radice 32 e il sottoalbero destro con radice 121. Si dice che i nodi con etichette 32 e 121 sono *fratelli* e sono rispettivamente il figlio sinistro e il figlio destro del nodo con etichetta 104. L'albero che ha come radice 32 ha ancora due sottoalberi con radici 23 e 44, i quali non hanno figli o, in altre parole, hanno come figli alberi vuoti. L'albero con radice 121 non ha figlio sinistro e ha come figlio destro il sottoalbero con radice 200, il quale a sua volta ha come figlio sinistro il sottoalbero con radice 152 e non ha figlio destro. Il sottoalbero che ha come radice 152 non ha figli.

I nodi da cui non si dipartono altri sottoalberi (non vuoti) sono detti *nodi terminali* o *foglie*. Nell'esempio sono quelli etichettati con: 23, 44 e 152.

Si chiamano *visite* le scansioni dell'albero che portano a percorrerne i vari nodi. L'ordine in cui questo avviene distingue differenti tipi di visite.

La visita *in ordine anticipato* di un albero binario viene effettuata con il seguente algoritmo:

```
anticipato( radice dell'albero )
     Se l'albero non è vuoto:
        Visita la radice
        anticipato( radice del sottoalbero sinistro )
        anticipato( radice del sottoalbero destro )
```

in cui abbiamo evidenziato la ricorsività della visita. Nel caso dell'albero di Figura 15.1 la visita in ordine anticipato dà la sequenza: 104, 32, 23, 44, 121, 200, 152.

La visita *in ordine differito* invece è così descritta:

```
differito( radice dell'albero )
     Se l'albero non è vuoto:
        differito( radice del sottoalbero sinistro )
        differito( radice del sottoalbero destro )
        Visita la radice
```

Applicando il procedimento sull'albero di esempio abbiamo: 23, 44, 32, 152, 200, 121, 104.

Concludiamo con un terzo tipo di visita, *in ordine simmetrico*, il cui algoritmo è:

```
simmetrico( radice dell'albero )
     Se l'albero non è vuoto:
        simmetrico( radice del sottoalbero sinistro )
        Visita la radice
        simmetrico( radice del sottoalbero destro )
```

che restituisce, con riferimento all'esempio di Figura 15.1: 23, 32, 44, 104, 121, 152, 200.

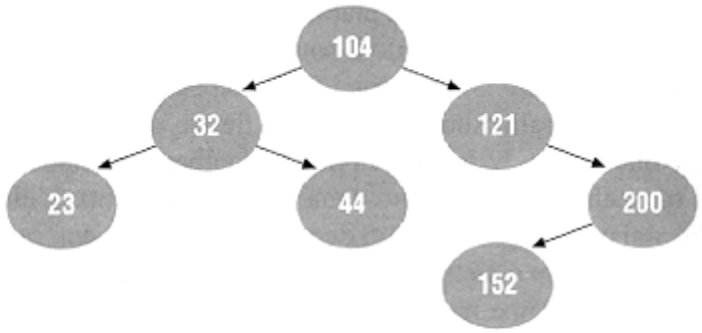


Figura 15.1 Esempio di albero binario

15.2 Implementazione di alberi binari

Consideriamo ora il seguente problema: memorizzare i dati interi immessi dall'utente in un albero binario tale che il valore dell'etichetta di un qualsiasi nodo sia maggiore di tutti i valori presenti nel suo sottoalbero sinistro e minore di tutti i valori del suo sottoalbero destro. Per esempio, se l'utente immette in sequenza 104, 32, 44, 121, 200 152, 23, un albero conforme con la definizione è quello di Figura 15.1. La sequenza di immissione termina quando l'utente inserisce il valore zero; in caso di immissione di più occorrenze dello stesso valore, soltanto la prima verrà inserita nell'albero. Si chiede di visitare l'albero in ordine anticipato.

Una soluzione è ottenuta definendo ciascun nodo come una struttura costituita da un campo informazione, contenente l'etichetta, e due puntatori al sottoalbero sinistro e al sottoalbero destro:

```
struct nodo {
       int inf;
       struct nodo *alb_sin;
       struct nodo *alb_des;
     };
```

L'albero binario di Figura 15.1 verrebbe così memorizzato come in Figura 15.2. I puntatori che non fanno riferimento a nessun nodo devono essere messi a valore `NULL`, in modo da permettere una corretta terminazione delle visite. Il programma completo è presentato nel Listato 15.1.

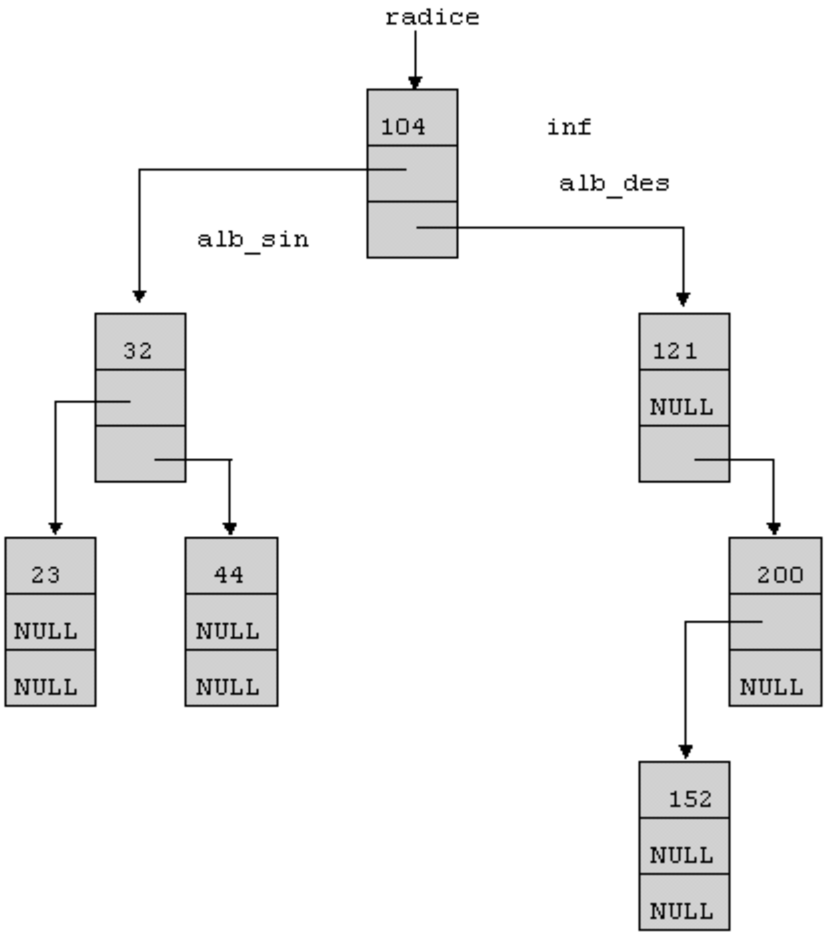


Figura 15.2 Esempio di memorizzazione dell'albero binario di Figura 15.1 in una lista doppia

Nel `main` è dichiarata la variabile `radice` che conterrà il riferimento alla radice dell'albero; essa è un puntatore a oggetti di tipo `nodo`:

```
struct nodo *radice;
```

Il `main` invoca la funzione `alb_bin` che crea l'albero e ritorna il puntatore alla radice:

```
radice = alb_bin();
```

Successivamente chiama la funzione di visita, che visualizza le etichette in ordine anticipato:

```
anticipato(radice);
```

Ad `anticipato` viene passato il puntatore alla radice dell'albero.

Listato 15.1 - Creazione e visita in ordine anticipato di una albero binario

La funzione `alb_bin` crea l'albero vuoto inizializzando a `NULL` il puntatore alla radice `p`. Il resto della procedura è costituito da un ciclo in cui si richiedono all'utente i valori della sequenza finché non viene immesso zero. A ogni nuovo inserimento viene chiamata la funzione `crea_nodo` la quale pensa a inserirlo nell'albero:

```
p = crea_nodo(p, x.inf);
```

A ogni chiamata `crea_nodo` restituisce la radice dell'albero stesso. La dichiarazione di `crea_nodo` è la seguente:

```
struct nodo *crea_nodo(struct nodo *p, int val);
```

Come abbiamo già visto per le liste, il caso della creazione del primo nodo deve essere trattato a parte. La funzione `alb_bin` ha provveduto a inizializzare la radice a valore `NULL`; in questo modo un test su `p` in `crea_nodo` dirà se

l'albero è vuoto. In questo caso verrà creato il nodo radice e inizializzato il campo `inf` con il valore immesso dall'utente, passato alla funzione nella variabile `val`:

```
/* Creazione del nodo */
     p = (struct nodo *) malloc(sizeof(struct nodo));
     p->inf = val;
     p->alb_sin = NULL;
     p->alb_des = NULL;
```

Prima di far ritornare il controllo ad `alb_bin` viene assegnato `NULL` ai puntatori sinistro e destro del nodo.
Nel caso esista almeno un nodo già costruito (`p` non è uguale a `NULL`), ci si domanda se `val` sia maggiore del campo `inf` della radice, nel qual caso viene richiamata ricorsivamente `crea_nodo` passandole il puntatore al sottoalbero destro:

```
p->alb_des = crea_nodo(p->alb_des, val);
```

Se al contrario `val` è minore di `inf`, allora viene richiamata ricorsivamente la funzione sul sottoalbero sinistro:

```
p->alb_sin = crea_nodo( p->alb_sin, val);
```

Se nessuno dei due casi risulta vero, significa che `val` è uguale a `inf` e dunque non deve essere inserito nell'albero perché, come specificava il testo del problema, le occorrenze multiple devono essere scartate.
Si noti come, nel caso di un valore non ancora memorizzato nell'albero, il procedimento ricorsivo termini sempre con la creazione di una foglia, corrispondente alle istruzioni di "creazione del nodo" che abbiamo elencato in precedenza. La connessione del nodo creato al padre avviene grazie al valore di ritorno delle chiamate ricorsive, che è assegnato a `p->alb_des` o a `p->alb_sin` secondo il caso.
La funzione `anticipato` corrisponde in maniera speculare alla definizione di visita in ordine anticipato data precedentemente. L'intestazione della definizione della funzione è la seguente:

```
void anticipato(struct nodo *p);
```

Il parametro attuale `p` assume il valore della radice, se non è uguale a `NULL` (cioè se l'albero non è vuoto), ne viene stampata l'etichetta, e viene invocata ricorsivamente `anticipato` passandole la radice del sottoalbero sinistro:

```
anticipato(p->alb_sin);
```

Successivamente viene richiamata la funzione passandole la radice del sottoalbero destro:
anticipato(p->alb_des);

15.3 Visita in ordine simmetrico

Il problema di questo paragrafo è: ampliare il programma del paragrafo precedente in modo che venga effettuata anche la visita in ordine simmetrico e inoltre sia ricercato nell'albero un valore richiesto all'utente. Nel Listato 15.2 sono riportate le modifiche da apportare al programma e le nuove funzioni.
La funzione di visita `simmetrico` è analoga ad `anticipato` esaminata precedentemente. La differenza sta nel fatto che prima viene visitato il sottoalbero sinistro, poi viene visitata la radice e infine viene visitato il sottoalbero destro.
La variabile `trovato` del `main` è di tipo puntatore a una struttura `nodo`:

```
struct nodo *trovato;
```

Essa viene passata per indirizzo alla funzione `ricerca` che vi immette, se reperito, il puntatore al nodo che contiene il valore ricercato. La funzione `ricerca` si comporta come la visita in ordine anticipato: prima verifica se l'elemento ricercato è nella posizione corrente, poi lo ricerca nel sottoalbero sinistro, infine lo ricerca nel sottoalbero destro.

/* Da aggiungere alle dichiarazioni iniziali del Listato 15.1 */

```
void simmetrico(struct nodo *);
void ricerca(struct nodo *, int, struct nodo **);
```

/* Da aggiungere al `main` del Listato 15.1 */

```
struct nodo *trovato;
int chi;
...
printf("\nVISITA IN ORDINE SIMMETRICO\n");
simmetrico(radice);

printf("\nInserisci il valore da ricercare: ");
scanf("%d", &chi);

printf("\nRICERCA COMPLETA");
trovato = NULL;
ricerca(radice, chi, &trovato);
if(trovato != NULL)
  printf("\n Elemento %d presente \n", trovato->inf);
else
  printf("\n Elemento non presente\n");


/* Funzione che visita l'albero binario in ordine simmetrico.
   Da aggiungere al Listato 15.1                             */

void simmetrico(struct nodo *p)
{
if(p!=NULL) {
  simmetrico(p->alb_sin);
  printf("%d  ", p->inf);
  simmetrico(p->alb_des);
}
}


/* Funzione che ricerca un'etichetta nell'albero binario.
   Da aggiungere al Listato 15.1.
   Visita l'albero in ordine anticipato                   */

void ricerca(struct nodo *p, int val, struct nodo **p_ele)
{
if(p!=NULL)
  if(val == p->inf)      /* La ricerca ha dato esito positivo */
    *p_ele = p;          /* p_ele è passato per indirizzo
                            per cui l'assegnamento di p
                            avviene sul parametro attuale */
  else {
    ricerca(p->alb_sin, val, p_ele);
    ricerca(p->alb_des, val, p_ele);
}
}
```

Listato 15.2 Visita in ordine simmetrico e ricerca di un valore nell'albero binario

15.4 Alberi binari in ricerca

L'algoritmo di ricerca che abbiamo implementato non è dei più veloci. Infatti, per accorgersi che un valore non è presente si deve visitare l'intero albero, realizzando cioè quella che abbiamo definito nel Capitolo 5 come una ricerca completa. Osserviamo però che, preso un qualsiasi nodo, il suo valore è maggiore di tutti i nodi presenti nel suo sottoalbero sinistro e minore di tutti i nodi del suo sottoalbero destro. Una ricerca che utilizzi questa informazione porta più velocemente al risultato (si veda il Listato 15.3).

Se il valore ricercato non è quello presente nel nodo attuale lo andiamo a ricercare nel sottoalbero sinistro se risulta esserne maggiore, altrimenti nel suo sottoalbero destro. Considerando l'albero di Figura 15.1 la ricerca del valore 23 o 200 si conclude in tre chiamate; se ricerchiamo un valore non presente la ricerca si chiude al massimo in cinque chiamate (il numero di livelli più uno).
In pratica abbiamo realizzato un algoritmo che ha prestazioni analoghe a quello di ricerca binaria. L'idea che sta dietro a questo modo di procedere è utilizzata per creare gli indici dei file.

```
/* Ricerca ottimizzata */

void ric_bin(struct nodo *p, int val, struct nodo **p_ele)
{
if(p!=NULL)
  if(val == p->inf) {
    printf("  trovato ");
    *p_ele = p;
  }
else
  if(val < p->inf) {
    printf("  sinistra");
    ric_bin(p->alb_sin, val, p_ele);
  }
  else {
    printf(" destra");
    ric_bin(p->alb_des, val, p_ele);
  }
}
```

Listato 15.3 Ricerca di un valore nell'albero binario

✓ **NOTA**
Osserviamo che la ricerca ha le stesse prestazioni di quella binaria solamente se l'albero è bilanciato, cioè se, preso un qualsiasi nodo, il numero di nodi dei suoi due sottoalberi differisce al più di una unità. Il numero di livelli dell'albero, infatti, costituisce un limite superiore al numero di accessi effettuati dalla ricerca.
La struttura dell'albero creato con la funzione `crea_nodo` dipende dalla sequenza con cui vengono immessi i valori in ingresso; il caso peggiore si presenta quando questi sono già ordinati. Per esempio, se l'utente inserisse: 23, 32, 44, 104, 121, 152, 200 l'albero si presenterebbe totalmente sbilanciato, un albero cioè dove il numero di livelli è uguale al numero di elementi inseriti. Altre sequenze danno vita ad alberi più bilanciati.
Si potrebbero scrivere algoritmi che bilanciano l'albero mentre lo creano, ma spesso nella realtà – specialmente se si sta lavorando con la memoria secondaria – questo risulta pesante in termini di tempi di risposta. Un'altra soluzione è quella di prevedere funzioni che trasformino alberi qualsiasi in alberi bilanciati, per cui l'inserimento di elementi avviene così come l'abbiamo visto e, a tempi determinati, viene lanciata la procedura di bilanciamento.

15.5 Alberi ordinati

Un nodo di un albero binario può avere zero, uno o due figli. Viene spontaneo pensare a una struttura dati che superi questa restrizione: dato un insieme di uno o più nodi A, un albero è così definito:

- • un nodo di A è scelto come radice;
- • i rimanenti nodi, se esistono, possono essere ripartiti negli insiemi disgiunti A_1, A_2, ..., A_n, essi stessi definiti come alberi.

Si noti comunque che la definizione data non include l'albero vuoto, come nel caso dell'albero binario; infatti si parte dall'ipotesi che A contenga uno o più nodi. Di nuovo siamo in presenza di una definizione di tipo ricorsivo: i sottoalberi

A_1, A_2, ..., A_n sono a loro volta alberi. Di solito si è interessati ad *alberi ordinati*, dove, oltre alla gerarchia insita nella struttura stessa, esiste un ordinamento tra i nodi presenti a ciascun livello dell'albero (Figura 15.3).

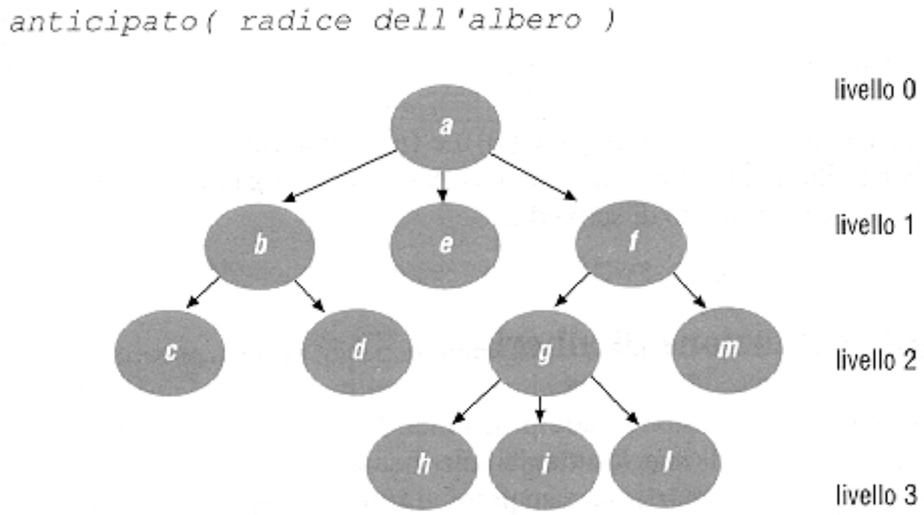


Figura 15.3 Rappresentazione di un albero

Nel caso si definisca l'ordinamento da sinistra verso destra, al livello 2 dell'albero della figura avremmo in sequenza: *c*, *d*, *g*, *m*. Analogamente al caso dell'albero binario, possiamo definire le visite per l'albero ordinato.
La visita in ordine anticipato è descritta da:

```
anticipato( radice dell'albero )
        Visita la radice
        fintantoché ci sono sottoalberi
           anticipato( radice del prossimo sottoalbero )
```

Nel caso dell'albero di Figura 15.3 avremmo: *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *l*, *m*.

La visita in ordine differito è descritta invece da:

```
differito( radice dell'albero )
        fintantoché ci sono sottoalberi
           differito( radice del prossimo sottoalbero )
        Visita la radice
```

Sempre con riferimento all'esempio si avrebbe: *c*, *d*, *b*, *e*, *h*, *i*, *l*, *g*, *m*, *f*, *a*.

Per rappresentare un albero in forma compatta si può utilizzare la rappresentazione parentetica in ordine anticipato, che ha la seguente sintassi:

```
(radice(I sottoalbero)(II sottoalbero) ... (N-esimo sottoalbero))
```

che, se si vuole, è un altro modo per esprimere la visita in ordine anticipato. Naturalmente i sottoalberi sono a loro volta alberi e si può applicare su di essi la stessa rappresentazione. Nel caso dell'esempio di Figura 15.3 abbiamo:

```
(a(albero con radice b)(albero con radice e)(albero con radice f))
```

Applicando nuovamente la stessa regola:

```
(a(b(c)(d))(e)(f(albero con radice g)(albero con radice m)))
```

fino ad arrivare alla rappresentazione completa:

```
(a(b(c)(d))(e)(f(g(h)(i)(l))(m)))
```

Si provi a disegnare gli alberi (*z*(*b*)(*d*)(*e*)(*f*(*r*(*s*)))) e (*t*(*o*(*p*(*y*(*u*(*x*)(*f*(*m*))))))). Naturalmente esiste la possibilità di definire una rappresentazione analoga in ordine differito. Lasciamo al lettore il compito di scriverla.

15.6 Implementazione di alberi

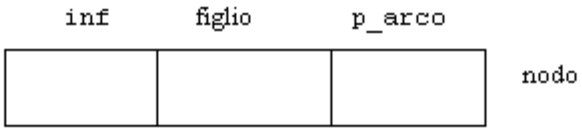
Per memorizzare gli alberi esistono varie possibilità. Una prima opzione potrebbe consistere nella definizione di una struttura simile a quella usata per l'albero binario, dove, oltre alla parte informazione, sono presenti tanti campi puntatore quanti ne sono necessari per far riferimento alle radici dei sottoalberi del nodo che ne ha di più:

```
struct nodo {
       char inf;
       struct nodo *p_1sott_albero;
       struct nodo *p_2sott_albero;
       ...
       struct nodo *p_nsotto_albero;
     };
```

Questa ipotesi presuppone la conoscenza del massimo numero di archi che si possono staccare da un nodo, e comporta inoltre uno spreco di memoria.

Un'altra possibilità è utilizzare liste multiple, in cui per ogni nodo dell'albero si forma una lista: il primo elemento della lista contiene l'etichetta del nodo, gli elementi successivi sono in numero uguale ai nodi figli. In ciascuno di essi viene memorizzato il puntatore alle liste di tali nodi.

```
struct nodo {
   char inf;
   struct nodo *figlio;
   struct nodo *p_arco;
};
```



Il campo `inf` viene utilizzato soltanto per il primo degli elementi di ogni lista, in cui il campo `p_arco` punta al secondo elemento della lista e `figlio` non viene utilizzato. Per gli altri elementi della lista `figlio` viene utilizzato per far riferimento alla lista del figlio e `p_arco` per puntare al successivo elemento della lista. In Figura 15.4 viene presentata la memorizzazione dell'albero di Figura 15.3.

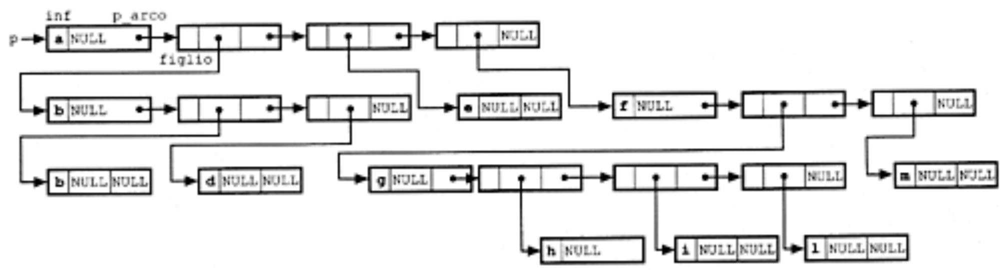


Figura 15.4 Memorizzazione dell'albero di Figura 15.3

Naturalmente questa scelta comporta un certo spreco di memoria, poiché soltanto due dei tre campi vengono utilizzati da ogni elemento allocato in memoria. Una soluzione alternativa è utilizzare due tipi di strutture: uno per il primo nodo di ogni lista, composta da un campo `inf` per l'etichetta del nodo e da un campo puntatore al secondo elemento della lista; un secondo per gli elementi successivi di ogni lista, composta da un campo puntatore alla lista di ogni figlio e da un campo puntatore al successivo elemento della lista:

```
struct nodo {
       char inf;
       struct lista *lista_figli;
     };

struct lista {
       struct nodo  *figlio;
       struct lista *successore;
     };
```

Per semplificare gli algoritmi di creazione e visita scegliamo di usufruire di un solo tipo di elemento.

Consideriamo ora il problema di memorizzare un albero immesso dall'utente in forma parentetica anticipata in una lista multipla e visitare in ordine anticipato l'albero costruito.

Nel Listato 15.4 è riportato il programma completo. Supponiamo per esempio che la stringa in ingresso sia quella relativa all'albero di Figura 15.3. La funzione `albero` legge il primo carattere della stringa in ingresso, che

necessariamente è una parentesi tonda aperta e chiama `crea_albero`. La funzione `crea_albero` alloca spazio per il primo nodo, legge il secondo carattere immesso e lo inserisce.
Viene poi letto un ulteriore carattere. Successivamente inizia un ciclo che va avanti finché il carattere in esame è uguale a parentesi tonda aperta, in pratica finché ci sono sottoalberi. Se l'albero fosse composto solamente dalla radice, a questo punto il procedimento avrebbe termine con il bloccaggio della lista inserendo `NULL` nel campo `p_arco` dell'elemento creato.
Nel ciclo viene allocato spazio per un successivo elemento, che questa volta verrà utilizzato per puntare al figlio, e viene chiamata ricorsivamente `crea_albero` passandole in ingresso il campo figlio dell'elemento creato. La nuova chiamata della funzione legge il successivo carattere della stringa, che necessariamente è la radice del sottoalbero, e così via finché ci sono figli.

```
/* Creazione di un albero e visita in ordine anticipato.
   L'albero viene immesso dall'utente informa parentetica
   anticipata. L'etichetta dei nodi è un carattere.
   L'albero è implementato con liste multiple             */

#include <stdio.h>
#include<stddef.h>

struct nodo {
  char inf;
  struct nodo *figlio;
  struct nodo *p_arco;
};

struct nodo *albero();
struct nodo *crea_albero(struct nodo *);
void anticipato(struct nodo *);

main()
{
struct nodo *radice;
radice = albero();                /* Creazione dell'albero */
printf("\nVISITA IN ORDINE ANTICIPATO\n");
anticipato(radice);
}


/* Legge il primo carattere della rappresentazione parentetica
   e invoca la funzione crea_albero()                        */
struct nodo *albero()
{
struct nodo *p = NULL;
char car;

printf("\nInserisci la rappresentazione dell'albero: ");
scanf("%c", &car);
p = crea_albero(p);
return(p);   /* Ritorna il puntatore alla radice al chiamante */
}


/* Crea un nodo e vi inserisce la relativa etichetta. Per ogni figlio crea un
arco. Invoca ricorsivamente se stessa          */
struct nodo *crea_albero(struct nodo *p)
{
struct nodo *paus;
char car;
```

```
/* crea il nodo */
p = (struct nodo *) malloc( sizeof( struct nodo ) );
scanf("%c", &p->inf);       /* Inserimento del valore nel nodo */
paus = p;
scanf("%c", &car);          /* Lettura del carattere successivo */

while(car=='(') {           /* Per ogni figlio ripeti */
  /* crea l'arco */
  paus->p_arco = (struct nodo *) malloc(sizeof(struct nodo));
  paus = paus->p_arco;
 /* Invoca se stessa passando il campo figlio dell'elem. creato */
  paus->figlio = crea_albero(paus->figlio);

  scanf("%c", &car);        /* Lettura del carattere successivo */
}

paus->p_arco = NULL;
return( p );   /* Ritorna il puntatore alla radice al chiamante */
}


/* Visita ricorsivamente l'albero in ordine anticipato */
void anticipato(struct nodo *p)
{
printf("(%c", p->inf);
p = p->p_arco;

while(p!=NULL) {
  anticipato(p->figlio);
  p = p->p_arco;
}
printf(")");
}
```

Listato 15.4 Creazione di un albero a partire dalla sua rappresentazione parentetica

15.7 Ricerca di un sottoalbero

Supponiamo di dover affrontare un ulteriore problema: dato l'albero allocato col procedimento del paragrafo precedente, ricercare il valore di una etichetta e, se presente, visitare il sottoalbero che da essa si diparte.
Considerando l'albero di Figura 15.3, se l'utente immette *f*, il programma deve visualizzare *f*, *g*, *h*, *i*, *l*, *m*. L'algoritmo di ricerca del Listato 15.5 si comporta come la visita in ordine anticipato, con l'eccezione che se, il valore viene reperito, il procedimento si blocca e ritorna il puntatore al nodo in esame.

/* Da aggiungere al Listato 15.4 /*

```
...
struct nodo *trovato;
char sotto_albero, invio;
...

scanf("%c", &invio);
printf("\nInserisci la radice del sottoalbero: ");
scanf("%c", &sotto_albero);

trovato = NULL;
ricerca( radice, sotto_albero, &trovato );
```

```
if(trovato!=NULL) {
  printf("\n SOTTOALBERO IN ORDINE ANTICIPATO \n");
  anticipato( trovato );
}
else
  printf("\n Sottoalbero non presente");
}


/* Visita in ordine anticipato, ricercando il sottoalbero con
radice sa. Se lo reperisce assegna il suo indirizzo a *punt_sa */

ricerca(struct nodo *p, char sa, struct nodo **punt_sa)
{
if(sa == p->inf)
  *punt_sa = p;
else
{
  p = p->p_arco;
  while(p!=NULL) {
    ricerca(p->figlio, sa, punt_sa);
    p = p->p_arco;
  }
}
}
```

Listato 15.5 Ricerca di un nodo e visualizzazione del sottoalbero che si diparte da esso

15.8 Trasformazione di alberi

Da un albero ordinato A di *n* nodi è possibile ricavare un equivalente albero binario B di *n* nodi con la seguente regola:

- • la radice di A coincide con la radice di B;
- • ogni nodo *b* di B ha come radice del sottoalbero sinistro il primo figlio di *b* in A e come sottoalbero destro il fratello successivo di *b* in A.

In Figura 15.5 vediamo la trasformazione dell'albero di Figura 15.3 in albero binario.

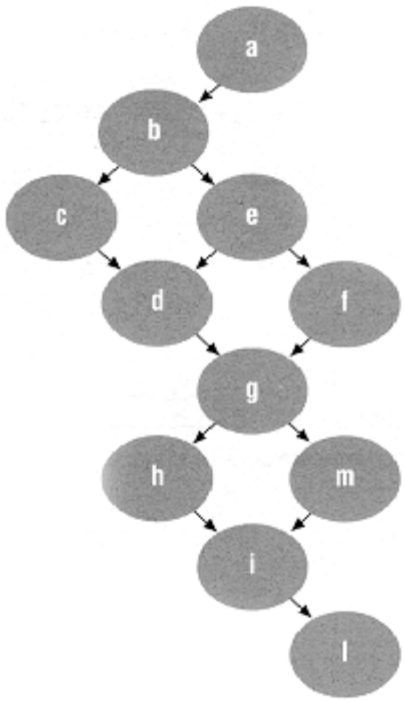


Figura 15.5 Albero binario corrispondente all'albero di Figura 15.3

Dalla regola di trasformazione si deduce che in un albero binario, ricavato da un albero ordinato, la radice può avere solamente il figlio sinistro.

Il programma del Listato 15.6 accetta in ingresso la rappresentazione parentetica di un albero ordinato e memorizza il corrispondente albero binario nella forma che abbiamo visto nei primi paragrafi di questo capitolo. Le funzioni utilizzate sono di tipo non ricorsivo e pertanto esiste il problema di memorizzare i puntatori degli elementi di livello superiore. A questo scopo si utilizza una pila di puntatori, dove si va a inserire il puntatore di un certo livello prima di scendere al livello sottostante e da dove si preleva il puntatore una volta che si debba risalire di livello.
La pila è implementata mediante una lista. Rimangono dunque valide le funzioni di `inserimento`, `eliminazione` e `pila_vuota` già trattate. L'unica differenza sta nella struttura di ogni elemento: la parte informazione è di tipo puntatore a un nodo dell'albero:

```
struct elemento {              /* Elemento della lista lineare  */
  struct nodo     *inf;        /* con cui è implementata la pila */
  struct elemento *pun;
};
```

Le visite le abbiamo realizzate come al solito in forma ricorsiva. Si può verificare che la visita dell'albero di partenza in ordine anticipato è uguale alla visita in ordine anticipato del corrispondente albero binario. La visita in ordine differito corrisponde invece alla visita in ordine simmetrico dell'albero binario.

```
/* Dalla rappresentazione parentetica di un albero ricava il
   corrispondente albero binario, che visita in ordine simmetrico,
   anticipato e differito.
   Per la creazione usa una funzione iterativa (non ricorsiva)
   con l'ausilio di una pila gestita mediante una lista lineare
   il cui campo inf è un puntatore ai nodi dell'albero binario.
   Per le  visite in ordine simmetrico, anticipato e differito
   rimangono valide le funzioni ricorsive già esaminate         */

#include <stdio.h>
#include <stddef.h>
```

```
struct nodo {                   /* Nodo dell'albero binario */
  char inf;
  struct nodo *alb_sin;
  struct nodo *alb_des;
};

struct nodo *alb_bin_par();    /* Funzione per la creazione */

void anticipato(struct nodo *);         /* Visite */
void simmetrico(struct nodo *);
void differito(struct nodo *);


struct elemento {              /* Elemento della lista lineare  */
  struct nodo     *inf;        /* con cui è implementata la pila */
  struct elemento *pun;
};

/* Funzioni per la gestione della pila */
struct elemento *inserimento(struct elemento *, struct nodo *);
struct elemento *eliminazione( struct elemento *, struct nodo **);
int pila_vuota(struct elemento *);

main()
{
struct nodo *radice;

radice = alb_bin_par();

printf("\nVISITA IN ORDINE SIMMETRICO\n");
simmetrico( radice );
printf("\nVISITA IN ORDINE ANTICIPATO\n");
anticipato( radice );
printf("\nVISITA IN ORDINE DIFFERITO\n");
differito( radice );
}


/* Dalla rappresentazione parentetica di un albero crea
   il corrispondente albero binario                      */
struct nodo *alb_bin_par()
{
struct nodo *p;
struct nodo *paus, *pp;
char car;
int logica = 1;

struct elemento *punt_testa = NULL;    /* Inizializzazione pila */

printf("\nInserisci la rappresentazione dell'albero: ");
scanf("%c", &car);

/* creazione della radice */
p = (struct nodo *) malloc(sizeof(struct nodo));

scanf("%c", &p->inf);

p->alb_sin = NULL;     /* Inizializzazione dei puntatori */
p->alb_des = NULL;     /* ai sottoalberi */
```

```
paus = p;  logica = 1;

do {
  scanf("%c", &car);
  if(car=='(') {
    pp = (struct nodo *) malloc(sizeof(struct nodo));
    scanf("%c", &pp->inf);
    pp->alb_sin = NULL;
    pp->alb_des = NULL;
    if(logica) {
      paus->alb_sin = pp;
      /* Inserimento in pila */
      punt_testa = inserimento(punt_testa, paus);
    }
    else {
      paus->alb_des = pp;
      logica = 1;
    }
    paus = pp;
   }
   else
     if(logica)
       logica = 0;
     else
       /* Eliminazione dalla pila */
       punt_testa = eliminazione(punt_testa, &paus);
}
while(!pila_vuota(punt_testa));

return(p);
}


/* Funzioni per la gestione della pila */

struct elemento *inserimento(struct elemento *p, struct nodo *ele)
{
struct elemento *paus;

paus = (struct elemento *)malloc(sizeof(struct elemento));
if(paus==NULL) return(NULL);

paus->pun = p;
p = paus;
p->inf = ele;

return(p);
}


struct elemento *eliminazione(struct elemento *p, struct nodo **ele)
{
struct elemento *paus;

*ele = p->inf;
paus = p;
p = p->pun;
free(paus);
```

```
return(p);
}


int pila_vuota(struct elemento *p)
{
if(p ==NULL)
  return(1);
else
  return(0);
}
```

/* Devono essere incluse le funzioni ricorsive di visita dell'albero binario già esaminate, dove il valore visualizzato è di tipo char /*

```
...
```

Listato 15.6 Creazione di un albero binario corrispondente all'albero ordinato (non binario) immesso in forma parentetica dall'utente

15.9 Grafi

Un *grafo* è costituito da un insieme di nodi e un insieme di archi che uniscono coppie di nodi tali che non c'è mai più di un arco che unisce una coppia di nodi. Il grafo è *orientato* quando viene attribuito un senso di percorrenza agli archi stessi: in tal caso è ammesso che tra due nodi vi siano due archi purché orientati in sensi opposti; è anche possibile che un arco ricada sullo stesso nodo (Figura 15.6). È evidente come gli alberi siano un caso particolare di grafi, in quanto rispondono a un insieme di regole più restrittivo.

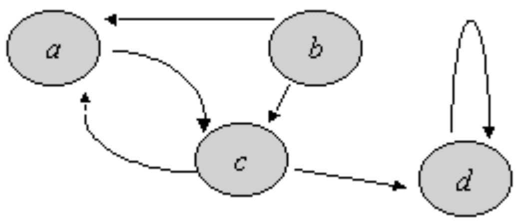


Figura 15.6 Rappresentazione di un grafo

Un modo per memorizzare i grafi, orientati e non, è quello di utilizzare una *matrice di adiacenza*, in cui ogni riga e ogni colonna rappresentano un nodo. Il grafo di Figura 15.6 viene memorizzato in una matrice di adiacenze come in Figura 15.7.

	a	*b*	*c*	*d*
a	0	0	1	0
b	1	0	1	0
c	1	0	0	1
d	0	0	0	1

Figura 15.7 Rappresentazione del grafo di Figura 15.6 con una matrice di adiacenze

Nel caso di grafi orientati la regola è la seguente: nella matrice appare un 1 se esiste un arco orientato dal nodo di riga al nodo di colonna. In particolare, si osservi il nodo *d*, che ha un arco orientato su se stesso.
La matrice di adiacenze provoca un notevole spreco di memoria nel caso, per esempio, che il numero *n* di nodi sia molto elevato in confronto al numero di archi: in ogni caso, infatti, si deve prevedere una matrice $n \times n$. Questa soluzione non è dunque molto flessibile.
Un'altra possibilità è rappresentata dalle liste di successori. In un array vengono memorizzati tutti i nodi e per ogni nodo è presente un puntatore a una lista che contiene i riferimenti ai nodi successori. In Figura 15.8 è presentato il grafo di Figura 15.6 memorizzato con liste di successori.

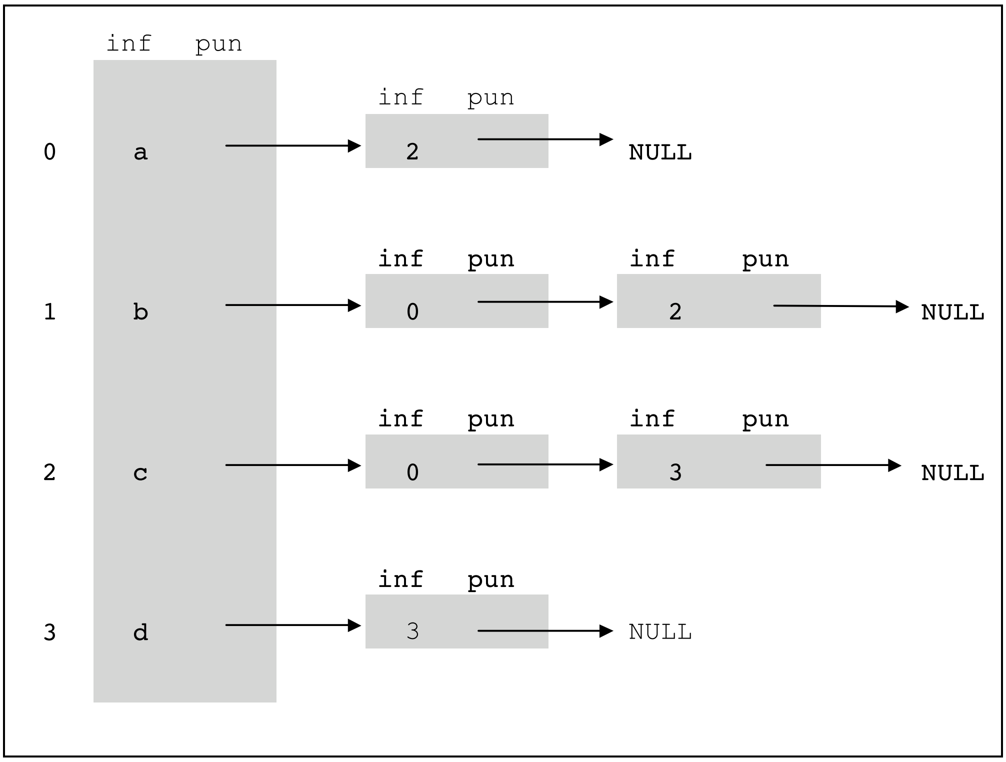


Figura 15.8 Grafo memorizzato mediante una lista di successori

Nel Listato 15.7 è presentato un programma che memorizza un grafo con una matrice di adiacenze, da cui ricava la rappresentazione in una lista di successori.
La funzione `mat_adiacenze` richiede all'utente i nodi e gli archi che conformano il grafo, memorizza le etichette dei nodi nel campo `inf` dell'array `s` e crea la matrice di adiacenze; successivamente la funzione `vis_mat_adiacenze` visualizza la matrice. La funzione `successori` percorre la matrice e per ogni suo elemento con valore 1 chiama la funzione `crea_succ`:

```
void crea_succ(int, int);
```

Questa funzione crea per ogni arco un elemento di lista inserendo nel campo informazione il riferimento al nodo di arrivo. La funzione `visita` percorre le liste di successori visualizzando tutti i nodi raggiungibili da un certo nodo.

✓ **NOTA**

Questa volta si è ritenuto utile mostrare un programma che fa uso essenzialmente di variabili globali cui tutte le funzioni fanno riferimento. Sappiamo che in generale è meglio far uso del passaggio di parametri come abbiamo mostrato in tutto il testo, ma non è detto che in casi particolari un'altra scelta non sia da apprezzare.

Abbiamo visto alcune delle strutture dati più importanti e abbiamo avuto modo di sperimentare la loro implementazione in linguaggio C. Nei problemi reali, pur rimanendo valide le logiche mostrate in queste pagine, le cose sono più complesse.
Il campo informazione, che abbiamo detto contenere l'etichetta di un nodo dell'albero, del grafo o di un'altra struttura, può contenere un insieme di dati molto ampio o essere un riferimento a essi. Per esempio, si pensi a un archivio di dati in memoria secondaria contenente un considerevole numero di record: per poter accedere a essi in tempi soddisfacenti si ha la necessità di utilizzare degli indici che permettano, dato il valore di una chiave (quale il codice di un articolo del magazzino o il conto corrente di un utente della banca), di ottenere tutte le informazioni a essa correlate.
In molti sistemi gli indici vengono realizzati con degli alberi, che consentono di accedere alle chiavi e da queste – tramite puntatori – al relativo record dell'archivio. Sono organizzati ad albero i file indici del linguaggio Cobol e di molti gestori di basi di dati. Gli indici stessi sono immagazzinati in file in memoria secondaria e vengono caricati in memoria centrale al momento della loro utilizzazione.
I grafi sono spesso usati nella soluzione di problemi di ricerca operativa che implicano, per esempio, lo studio di cammini minimi.

```
/* Trasformazione della rappresentazione di un grafo
   da una matrice di adiacenze a una lista di successori */

#include <stdio.h>
#include <malloc.h>

struct nodo {               /* Struttura di un nodo */
  char inf;
  struct successore *pun;
};

struct successore {        /* Elemento della lista di successori */
  int inf;
  struct successore *pun;
};

int a[10][10];             /* Matrice di adiacenze */
struct nodo s[10];         /* Array di nodi */
int n;                     /* Numero di nodi */

void mat_adiacenze(void);
void vis_mat_adiacenze(void);
void successori(void);
void crea_succ(int, int);
void visita(void);

main()
{
mat_adiacenze();       /* Creazione della matrice di adiacenze */
vis_mat_adiacenze();   /* Visualizzazione della matrice */
successori();          /* Creazione delle liste di successori */
visita();              /* Visual. dei successori di ogni nodo */
}


/* Crea la matrice di adiacenze */

void mat_adiacenze(void)
{
```

```
int i, j;
char invio;

printf("\nNumero di nodi: ");
scanf("%d", &n);
scanf("%c", &invio);

for(i=0; i<n; i++) {              /* Richiesta etichette dei nodi */
  printf("\nEtichetta del nodo: ");
  scanf("%c", &s[i].inf);
  scanf("%c", &invio);
  s[i].pun = NULL;
}

for(i=0; i<n; i++)                /* Richiesta archi orientati */
  for(j=0; j<n; j++) {
    printf("\nArco orientato da [%c] a [%c] (0 no, 1 si) ? ",
            s[i].inf, s[j].inf);
    scanf("%d", &a[i][j]);
  }
}


/* Visualizza la matrice di adiacenze */

void vis_mat_adiacenze(void)
{
int i, j;

printf("\nMATRICE DI ADIACENZE\n");
for(i=0; i<n; i++)                 /* Visualizza i nodi (colonne) */
  printf("   %c", s[i].inf);

for(i=0; i<n; i++) {
  printf("\n%c  ", s[i].inf);     /* Visualizza i nodi (righe) */
  for(j=0; j<n; j++)
    printf("%d   ", a[i][j]);     /* Visualizza gli archi */
}
}


/* Crea le liste di successori. Per ogni arco rappresentato
   nella matrice di adiacenze chiama crea_succ()            */

void successori(void)
{
int i, j;

for(i=0; i<n; i++)
  for(j=0; j<n; j++) {
    if(a[i][j]==1)
      crea_succ(i, j);
   }
}


/* Dato un certo arco nella matrice di adiacenze crea
il rispettivo elemento di lista                          */
```

```
void crea_succ( int i, int j )
{
struct successore *p;

if(s[i].pun==NULL) {      /* Non esiste la lista dei successori */
  s[i].pun = (struct successore *)(malloc(sizeof(struct successore)));
  s[i].pun->inf = j;
  s[i].pun->pun = NULL;
}
else {                    /* Esiste la lista dei successori */
  p = s[i].pun;
  while(p->pun!=NULL)
    p = p->pun;
  p->pun = (struct successore *)(malloc(sizeof(struct successore)));
  p = p->pun;
  p->inf = j;
  p->pun = NULL;
}
}


/* Per ogni nodo del grafo restituisce i suoi successori.
   Lavora sulle liste di successori                      */

void visita(void)
{
int i;
struct successore *p;

printf("\n");

for(i=0; i<n; i++) {
  printf("\n[%c] ha come successori: ", s[i].inf);
  p = s[i].pun;
  while(p!=NULL) {
    printf(" %c", s[p->inf].inf);
    p = p->pun;
  }
}
}
```

Listato 15.4 Rappresentazione di grafi

15.10 Esercizi

* 1. Scrivere una funzione ricorsiva per visitare in ordine differito un albero binario. Si consideri la stessa struttura dei nodi vista nel primo esempio di questo capitolo.

* 2. Si consideri la seguente definizione di nodo di albero binario:

```
struct nodo {
       int inf;
       int occorrenze;
       struct nodo *alb_sin;
       struct nodo *alb_des;
```

```
      };
```

Costruire la funzione `crea_nodo2` modificando `crea_nodo` del primo esempio di questo capitolo, in modo che venga calcolato il numero di occorrenze multiple dello stesso valore, eventualmente immesse dall'utente, e vengano memorizzate nel nodo stesso nel campo `occorrenze`.

* 3. Modificare la funzione che visita in forma simmetrica l'albero binario in modo che visualizzi, oltre al valore di ogni nodo, il corrispondente numero di occorrenze, memorizzate mediante la funzione `crea_nodo2` dell'esercizio precedente.

* 4. Scrivere una funzione ricorsiva per visitare in ordine differito un albero implementato mediante liste multiple. Si consideri la stessa struttura dei nodi vista negli esempi di questo capitolo:

```
struct nodo {
       char inf;
       struct nodo *figlio;
       struct nodo *p_arco;
      };
```

5. Scrivere una funzione che calcoli il numero di livelli di un albero binario memorizzato in una lista doppia.

6. Scrivere un programma che da un albero binario, memorizzato in una lista doppia, elimini (rilasciando opportunamente la memoria) il sottoalbero la cui etichetta della radice viene passata in ingresso dall'utente.

* 7. Ampliare il programma di implementazione di un grafo mediante una lista di successori esaminato nel capitolo, in modo che accetti in ingresso il valore di un'etichetta e visualizzi le etichette di tutti i nodi da esso raggiungibili. Per la scansione dei nodi connessi si utilizzi una funzione ricorsiva.
[Prestare attenzione al fatto che la scansione può portare a un ciclo infinito se nel percorso si passa per più di una volta sullo stesso nodo.]

8. Scrivere una funzione che un dato grafo, memorizzato in una matrice di adiacenze, e date in ingresso le etichette di due nodi, verifichi se esiste un arco che li collega, nel qual caso lo cancelli.

9. Verificare se, dato un grafo memorizzato in una matrice di adiacenze, e date in ingresso le etichette di due nodi, dal primo nodo è possibile arrivare al secondo attraverso un cammino di archi orientati.

10. Risolvere i due esercizi precedenti ma con il grafo memorizzato in una lista di successori.

11. Disegnare l'albero binario che si ottiene fornendo al programma del Listato 15.1 i seguenti dati:
35, 12, 91, 7, 13, 108, 64, 66, 19, 12, 8, 0.

12. Scrivere una funzione che stabilisca se due alberi binari sono uguali.

13. Dato l'albero binario costruito con il Listato 15.1 realizzare due funzioni, la prima che calcoli la somma totale delle informazioni contenute nei nodi, l'altra che determini il maggiore e il minore.

14. Scrivere una funzione che calcoli il numero di livelli di un albero memorizzato in una lista multipla.

15. Dato un albero ordinato, realizzare un programma completo per la sua gestione, dando all'utente un menu che comprenda almeno le opzioni per l'inserimento di un valore, l'eliminazione, le visite, la visualizzazione delle sole foglie.

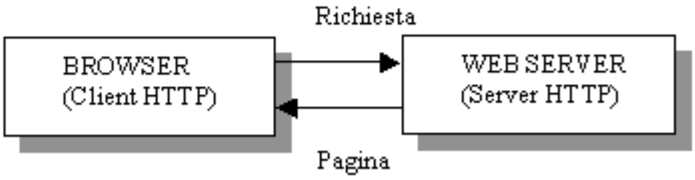
16.1 HTTP

Ora che abbiamo studiato e sperimentato il C e abbiamo risolto problemi di diversa natura e complessità, affrontiamo la realizzazione di applicazioni che abbiano come dominio Internet. Di fatto anche con il C possiamo fare tutto ciò che è possibile fare su Internet: inviare e ricevere posta elettronica, spedire e ricevere file, accedere a macchine remote, colloquiare con qualsiasi Web server. Ovviamente tutto ciò da programma.

*In questo contesto ci concentreremo su uno degli aspetti più interessanti: la programmazione dal lato del Web server, sfruttando il meccanismo delle cosiddette CGI (*Common Gateway

Interface*). Infatti, oltre a richiedere e ricevere pagine HTML (*HyperText Markup Language*), immagini, file audio e simili, è possibile anche innescare dal lato del server delle vere e proprie applicazioni che hanno la libertà di svolgere computazioni, accedere a basi di dati, effettuare connessioni con altri server e quanto altro è possibile fare da programma, per poi restituire al browser un risultato in forma di pagina HTML.*

Internet offre in forma di pagine informazioni e servizi. La scelta è davvero molto vasta, e va dalla semplice presentazione di annunci pubblicitari fino alla realizzazione di servizi complessi come la vendita o il noleggio di un'auto. Tutto questo avviene per mezzo di un meccanismo, concettualmente piuttosto semplice, illustrato in Figura 16.1, che va sotto il nome di protocollo HTTP (*HyperText Transfer Protocol*).



In pratica quando sul vostro browser – Netscape Navigator, Internet Explorer, Opera, Tango o altro – digitate un indirizzo Web, per esempio `www.mcgraw-hill.com`, inviate attraverso la rete Internet una richiesta al Web Server della McGraw-Hill, il quale risponderà inviandovi la prima pagina, detta *home page*, di quel sito. Volendo usare un linguaggio tecnicamente più preciso, il browser, che svolge la funzione di un client HTTP, invia una *HTTP request* al Web Server, che svolge il ruolo di HTTP Server, il quale a sua volta risponde con una *HTTP response*.

Il protocollo HTTP stabilisce anche il formato che devono avere sia la request sia la response. Per esempio una response, che corrisponde a una pagina scritta in linguaggio HTML, sarà costituita da una intestazione convenzionale, giusto per dire che quella è una response, cui segue il corpo vero e proprio della risposta in forma di pagina HTML. Vedremo più avanti i dettagli di questa operazione. Comunque, nella sua vera essenza, il protocollo HTTP e con esso tutta la "magia" della navigazione Internet consiste in questo: una richiesta seguita da una risposta e niente più. È una specie di protocollo di tipo "botta e risposta" che – più propriamente – è detto protocollo *stateless* o privo di memoria.

16.2 Pagine statiche

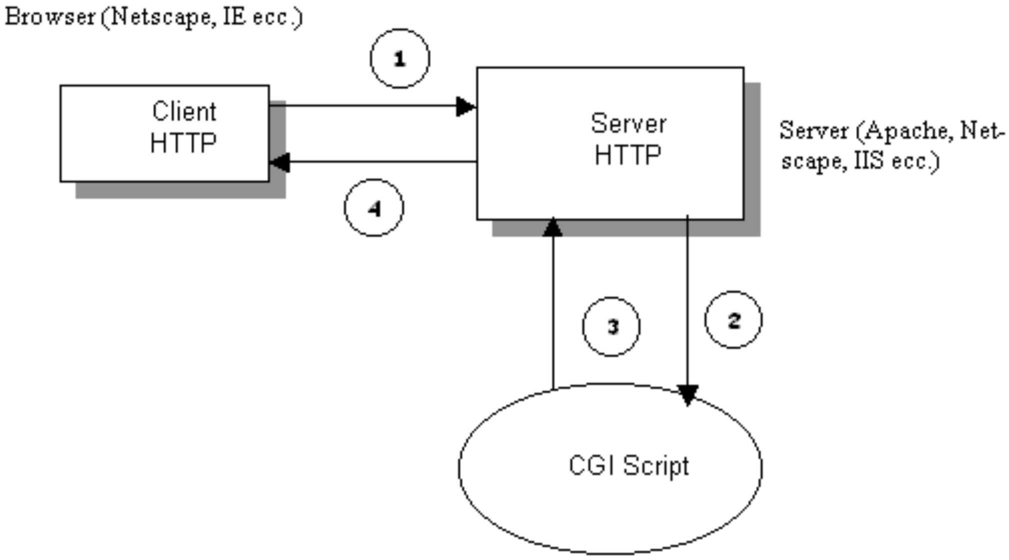
Quando il browser chiede al Web Server una pagina, per esempio per mezzo dell'indirizzo

```
www.cyberscuola/org/ilC.htm
```

e la pagina `ilC.htm` è disponibile come file sulla macchina che ospita il Web Server, questo non fa altro che leggere questa pagina, imbustarla in una response HTTP e inviarla indietro al browser. Pagine di questo tipo sono dette *pagine statiche*, poiché il server HTTP non effettua alcuna elaborazione se non quella di prendere la pagina così com'è e inviarla al browser.

16.3 Pagine dinamiche

Nel protocollo HTTP è anche possibile rispondere a un browser con una pagina non già disponibile sul disco ma costruita al volo, o – come si usa dire più correttamente –in modo dinamico, per mezzo di un programma. Il meccanismo con cui attraverso un server HTTP si lancia un programma, meccanismo detto dei CGI (*Common Gateway Interface*), estende e generalizza la semplice interazione request/response che abbiamo visto essere il cuore del protocollo HTTP (Figura 16.2).



`/cgi-bin/hello.exe`

Descriviamo ora passo per passo il meccanismo dei CGI prendendo a esempio un semplice programma – `hello.exe` – che costruisce una pagina dinamica HTML che presenta sul browser del richiedente il classico messaggio:

```
Hello, World!
```

1. Invio della request – Il client HTTP, ovvero un browser, effettua una request a un server HTTP identificato dal seguente indirizzo, detto URL:

`http://www.cyberscuola.org/cgi-bin/hello.exe?`

In questo indirizzoviene identificato il server HTTP

`http://www.cyberscuola.org`

e, relativamente a esso viene invocata una procedura CGI il cui riferimento è

`cgi-bin/hello.exe`

La directory `/cgi-bin` è una sottodirectory della directory dove è allocato il Web Server che contiene il programma CGI. Il nome del programma CGI è `hello.exe`.

2. Attivazione del CGI – Il server HTTP, che può essere basato su uno dei server attualmente disponibili, come Apache, Netscape Server o IIS di Microsoft, riceve la URL, la interpreta e lancia

un nuovo processo (o *thread*, a seconda del sistema) che esegue il CGI, ovvero il programma `hello.exe`.

3. Risposta del CGI – Il risultato della computazione deve comunque dare luogo a una pagina HTML di risposta che la procedura CGI invia verso il suo standard output, STDOUT, tenendo in parte conto anche di quello che deve essere il formato di una response HTTP. Lo STDOUT di ogni CGI non è però connesso al video, come si era visto nei capitoli precedenti, ma viene intercettato dal server HTTP che ha innescato il CGI.

4. Risposta del server HTTP – Penserà poi lo stesso server HTTP a inoltrare verso il client che aveva effettuato la request la pagina HTML – più precisamente la response HTTP – che aveva elaborato la procedura CGI.

Vediamo ora nel concreto come si innesca un CGI da HTML e come si scrive un CGI in C, a partire proprio dal semplice esempio `hello.exe`.

16.4 Un semplice CGI: `hello.exe`

Prima di descrivere e commentare il CGI `hello.exe` vediamo a livello di utente quali sono le azioni che si debbono intraprendere. Per prima cosa invocheremo da browser il CGI `hello.exe` (Figura 16.3): in risposta otterremo la pagina di saluto di Figura 16.4.



Figura 16.3 Invocazione del CGI `hello.exe`



Figura 16.4 Pagina di saluto ottenuta con l'esecuzione del CGI `hello.exe`

Visti gli effetti, vediamo ora quali sono state le "cause" di detti effetti nel Listato 16.1 e commentiamo questo semplice programma CGI.

```
/* hello.c: un esempio di CGI */

/* Includere ciò che serve per lo Standard Input e Output */
#include <stdio.h>

/* Dichiarare il main, come sempre */
int main(int argc, char *argv[])
{
/* Per prima cosa indicare un'informazione necessaria
per l'intestazione della response HTTP                    */
     printf("Content-type: text/html\n\n");
/* Inviare su standard Output i tag HTML */
     printf("<head>\n");
     printf("<title>Hello, World</title>\n");
     printf("</head>\n");
     printf("<body>\n");
     printf("<h1>Hello, World</h1>\n");
     printf("</body>\n");

     return 0;
}
```

Listato 16.1 CGI che produce il saluto "Hello World"

Il CGI è un programma lanciato dal server HTTP su richiesta del browser. Come programma dipendente dal server HTTP esso ha uno Standard Input e uno Standard Output che in apparenza funzionano, rispettivamente, come l'inserimento da tastiera e la visualizzazione sul monitor, ma in realtà sono due canali di comunicazione attraverso cui il CGI comunica con il server. Più specificatamente, il server HTTP invia eventuali parametri al CGI tramite STDIN, come vedremo più avanti, e riceve i risultati delle elaborazioni del CGI dallo STDOUT del CGI.

Dunque la prima cosa da fare è quella di includere le funzioni che permettono di gestire lo Standard Input e Output:

```
#include <stdio.h>
```

Dopo questa inclusione il programma procede come sempre per mezzo della dichiarazione della funzione `main`:

```
int main(int argc, char *argv[])
```

A questo punto comincia la costruzione "al volo" della pagina dinamica a partire da un pezzo di informazione relativo all'intestazione della response HTTP:

```
printf("Content-type: text/html\n\n");
                /* si notino le due righe vuote dopo
                   la scritta Content-type: text/html */
```

Come si può osservare, il programma invia su STDOUT, che – ricordiamo ancora una volta – viene intercettato dal server HTTP e poi inoltrato verso il browser, un messaggio che se visualizzato su monitor darebbe luogo alla scritta:

```
Content-type: text/html
```

Il significato di questo messaggio potrebbe essere così interpretato: "Il contenuto (*content-type*) di quello che sto inviando è un testo ASCII corrispondente alla specifica di una pagina HTML (`text/html`)". Si noti la presenza delle due righe vuote `\n\n`, assolutamente necessarie e, purtroppo, spesso dimenticate.
Infine viene trasmessa, rigo per rigo, la pagina HTML dinamica che realizza il citato messaggio, dapprima inviando su standard output i tag di inizio pagina (in Appendice A riportiamo una breve introduzione al linguaggio HTML che chiarisce il significato dei tag utilizzati nel testo):

```
printf("<head>\n");
printf("<title>Hello, World</title>\n");
printf("</head>\n");
```

poi procedendo con la specifica del corpo della pagina:

```
printf("<body>\n");
printf("<h1>Hello, World</h1>\n");
printf("</body>\n");
```

In effetti la versione statica equivalente di questa pagina dinamica sarebbe stata:

```
<head>
<title>Hello, World</title>
</head>
<body>
<h1>Hello, World</h1>
</body>
```

L'esempio, come abbiamo sottolineato, è molto semplice, ma ci è comunque servito per realizzare il nostro primo programma CGI

16.5 Cosa si può fare con un CGI?

Ora che abbiamo scoperto il meccanismo attraverso cui è possibile lanciare un programma su un Web Server, si dischiude un mondo di possibili applicazioni. Per esempio, se colleghiamo a una macchina Web Server un semaforo possiamo attraverso un programma CGI accendere e spegnere le luci del semaforo. Esempi di applicazioni di questo tipo, cioè di apparecchiature collegate a un Web Server che vengono comandate remotamente da browser innescando un CGI, sono numerose su Internet.
La tecnologia Web si sta sempre più rapidamente diffondendo anche per applicazioni più concrete. Elettrodomestici intelligenti, come frigoriferi cui è possibile chiedere se sono vuoti o no, i quali possono risponderci dicendo che cosa manca, non sono più fantascienza ma realtà. Per il momento, però, accontentiamoci di qualcosa di più modesto, ma che concettualmente è assimilabile alla gestione di apparati.
Illustriamo ora un CGI che legge l'ora esatta (o quasi) del Web Server e la trasmette in forma di pagina HTML al browser (Listato 16.2).

```
/* Ora esatta (o quasi) */
#include <stdio.h>
#include <time.h>

int main(int argc, char *argv[])
{
/* Si dichiarano due variabili per memorizzare l'ora */
time_t bintime;
struct tm *curtime;

     printf("Content-type: text/html\n\n");

     printf("<head>\n");
     printf("<title>Ora (quasi) Esatta</title>\n");
     printf("</head>\n");
     printf("<body>\n");
     printf("<h1>\n");
     time(&bintime);
     curtime = localtime(&bintime);
     printf("Ora (quasi) esatta: %s\n",asctime(curtime));
     printf("</h1>\n");
     printf("</body>\n");

     return 0;
}
```

Listato 16.2 CGI che trasmette l'ora esatta

Procediamo con i commenti. Per prima cosa occorre includere nel programma le funzioni che consentono la gestione dell'orologio di sistema:

```
#include <time.h>
```

Poi, subito dopo la dichiarazione del `main`, si dichiarano due variabili:

```
time_t bintime;
struct tm *curtime;
```

La prima variabile `bintime` è di tipo `time_t`. Il tipo `time_t` definito in `time.h` non è altro che un `long integer` destinato a memorizzare ora e data misurate in numero di secondi trascorsi dalla mezzanotte del 1° gennaio 1970, ora di Greenwich. Siamo d'accordo che questo è un modo piuttosto singolare di registrare ora e data, ma così è perché così si è misurato il tempo sui sistemi Unix. Se il lettore è interessato provi a calcolare quando un `long integer` non sarà più sufficiente e avremo, di conseguenza, un nuovo problema stile anno 2000.
La seconda variabile `curtime` è un puntatore a una struttura, anch'essa definita in `time.h`, che rappresenta ora e data in una forma un po' meno esotica:

```
struct tm {
  int tm_sec;     /* seconds after the minute - [0,59] */
  int tm_min;     /* minutes after the hour - [0,59] */
  int tm_hour;    /* hours since midnight - [0,23] */
  int tm_mday;    /* day of the month - [1,31] */
  int tm_mon;     /* months since January - [0,11] */
  int tm_year;    /* years since 1900 */
  int tm_wday;    /* days since Sunday - [0,6] */
  int tm_yday;    /* days since January 1 - [0,365] */
  int tm_isdst;   /* daylight savings time flag */
  };
```

Dopo aver costruito l'intestazione della response:

```
printf("Content-type: text/html\n\n");
```

provvediamo a costruire l'intestazione della pagina:

```
printf("<head>\n");
printf("<title>Ora (quasi) Esatta</title>\n");
printf("</head>\n");
```

Il corpo della pagina è stavolta leggermente più articolato rispetto al semplice `hello.exe`:

```
printf("<body>\n");
printf("<h1>\n");
time(&bintime);
curtime = localtime(&bintime);
printf("Ora (quasi) esatta: %s\n",asctime(curtime));
printf("</h1>\n");
printf("</body>\n");
```

La funzione:

```
time(&bintime);
```

legge ora e data in secondi dal 1° gennaio 1970. La funzione:

```
curtime = localtime(&bintime);
```

trasforma questi secondi in dati organizzati secondo la struttura `tm`, e la funzione

```
asctime(curtime)
```

trasforma il contenuto della struttura in una stringa ASCII. Il risultato di questo CGI potrebbe essere quello di Figura 16.5.

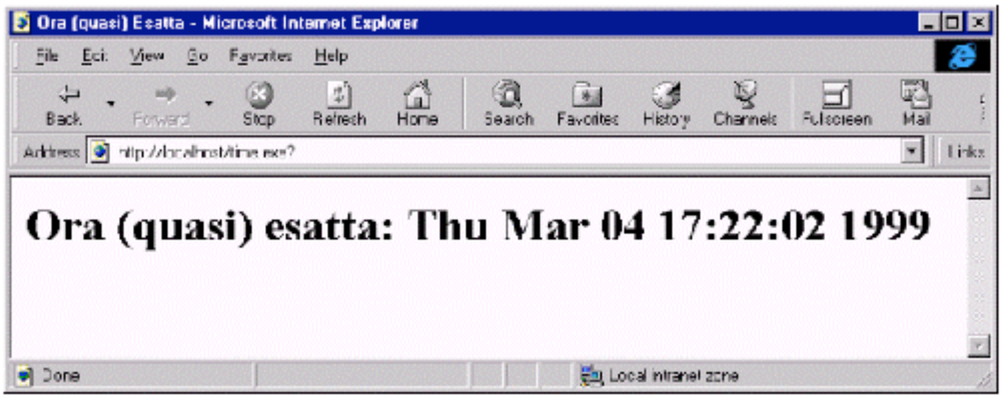


Figura 16.5 Pagina con l’ora esatta ottenuta con il CGI del Listato 16.2

16.6 Il passaggio di parametri

Ora che abbiamo acquisito una certa confidenza con la programmazione CGI affrontiamo un altro tema di interesse: come si effettua il passaggio di parametri a un CGI. Per prima cosa occorre dire che il modo canonico di invocare un CGI passando dei parametri è quello di impiegare un FORM HTML. Il FORM è un tag del linguaggio HTML che consente di costruire moduli per l’inserimento di informazioni. Vediamo subito un esempio in Figura 16.6. I campi vuoti di questo FORM possono essere riempiti come in Figura 16.7.



Figura 16.6 Esempio di FORM

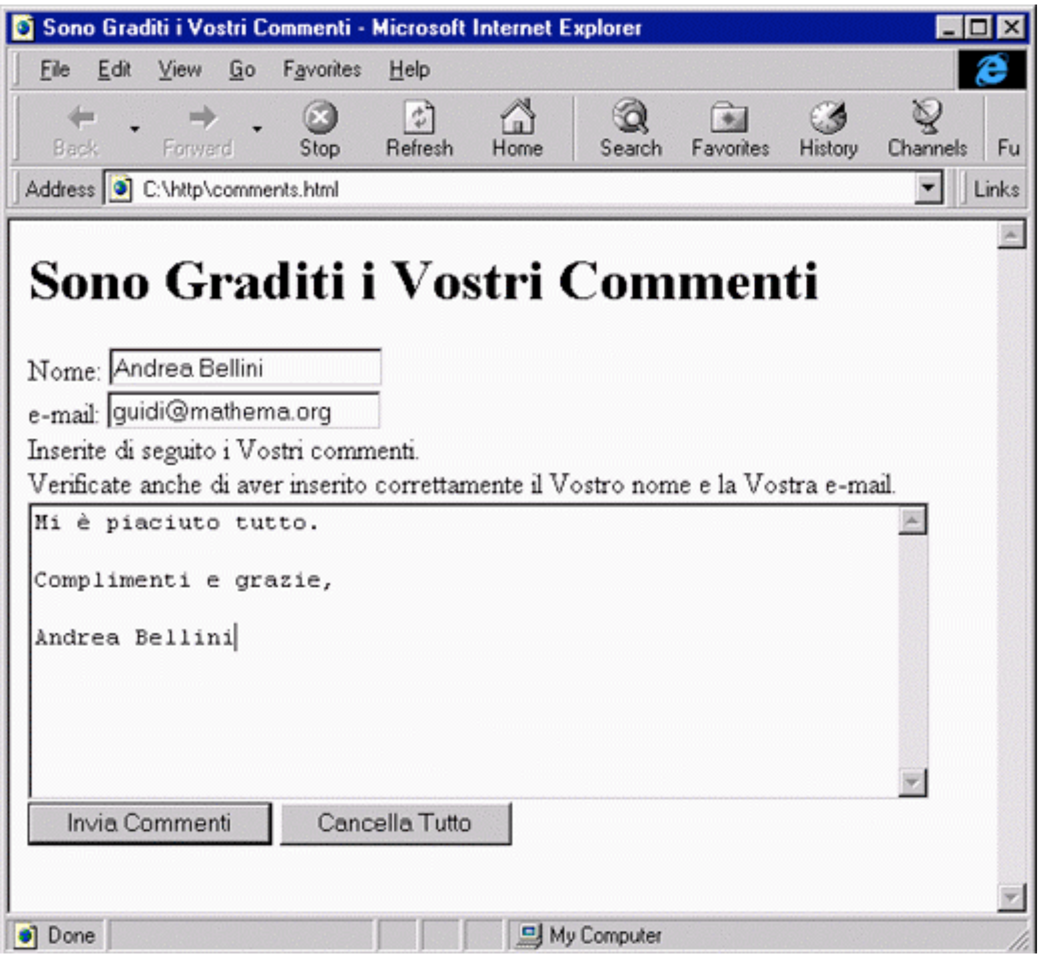


Figura 16.7 Esempio di FORM con i campi riempiti

Premendo il pulsante “Invia Commenti” si attiva il CGI che acquisisce i dati immessi, effettua delle elaborazioni e ritorna una pagina HTML per confermare l’avvenuta elaborazione. Il codice HTML che ha permesso di generare questa pagina è quello del Listato 16.3.

```
<html>
<head>
<title>Sono Graditi i Vostri Commenti</title>
</head>
<body>
<h1>Sono Graditi i Vostri Commenti</h1>
<p>
<form action="http://localhost/comments.exe" method="POST">
Nome: <input type="text" name="name"><br>
e-mail: <input type="text" name="email"><br>
Inserite di seguito i Vostri commenti.<br>
Verificate anche di aver inserito correttamente il Vostro nome e
la Vostra e-mail.<br>
<textarea name="comments" rows="10" cols="60">
</textarea>
<br>
<input type="submit" value="Invia Commenti">
<input type="reset" value="Cancella Tutto">
```

```
</form>
</body>
</html>
```

Listato 16.3 Codice HTML per la gestione della pagina delle Figure 16.6 e 16.7

La parte che ci interessa commentare è quella relativa al tag `<form>`:

```
<form action="http://localhost/comments.exe" method="POST">
Nome: <input type="text" name="name"><br>
e-mail: <input type="text" name="email"><br>
Inserite di seguito i Vostri commenti.<br>
Verificate anche di aver inserito correttamente il Vostro nome e
la Vostra e-mail.<br>
<textarea name="comments" rows="10" cols="60">
</textarea>
<br>
<input type="submit" value="Invia Commenti">
<input type="reset" value="Cancella Tutto">
</form>
```

Si osservi la prima direttiva:

```
<form action="http://localhost/comments.exe" method="POST">
```

In essa si inizia un FORM e come azione associata al FORM, ovvero come CGI da innescare in seguito alla selezione del bottone "Invia Commenti", lanceremo il CGI identificato da

```
http://localhost/comments.exe
```

facendo uso del metodo di passaggio di parametri detto "POST". Esistono due metodi per inviare i parametri a un CGI.

GET i parametri sono passati al CGI per mezzo della variabile di ambiente `QUERY_STRING`, e in questa variabile di solito non si possono memorizzare più di un centinaio di byte di dati;

POST i parametri vengono trasmessi al CGI per mezzo dello standard input del CGI senza alcuna limitazione di dimensione. In questo caso il CGI può conoscere quanti dati deve leggere da STDIN andando a leggere il contenuto di un'altra variabile di memoria `CONTENT_LENGTH`.

Il programma CGI per sapere quale metodo è stato scelto per passargli i parametri è sufficiente che legga il contenuto della variabile di ambiente `REQUEST_METHOD`. Da questa semplice introduzione si arguisce come il metodo POST sia tra i due preferibile poiché non introduce alcun limite al passaggio dei parametri. Ma in che modo vengono passati i parametri?
Ritorniamo alla definizione in HTML del FORM definito nell'esempio, a partire dalla definizione del primo campo relativo al "Nome":

```
Nome: <input type="text" name="name">
```

La prima parte

```
Nome:
```

non ha nessun significato speciale se non quello di essere una etichetta che esplicita il significato del campo; il tag

```
<input type="text" name="name">
```

invece definisce il campo da riempire. Il significato di questo tag potrebbe essere così parafrasato: “Definisci un campo di input, ovvero un campo in cui effettuare un inserimento. Il valore che sarà inserito nel campo sarà trattato come un testo (`type = "text"`) e il nome simbolico che daremo a questo campo sarà `"name"` (`name="name"`)”. Il nome simbolico del campo è molto importante. Infatti quando digiteremo qualcosa in questo campo, per esempio `"Andrea Bellini"`, nel momento in cui il FORM con i suoi contenuti inseriti dall’utente sarà inviato al CGI darà luogo a una coppia del tipo:

```
name=Andrea+Bellini
```

Si noti come lo spazio venga simbolicamente rappresentato con il simbolo '+'. Il successivo campo relativo all’indirizzo e-mail:

```
e-mail: <input type="text" name="email">
```

ha il medesimo significato del precedente, ma con un diverso nome simbolico, `"email"`. A questo punto la stringa di parametri che sarà passata al CGI si allunga:

```
name=Andrea+Bellini&email=guidi@mathema.com
```

Si osservi come le coppie

```
<nome=valore>
```

vengano separate per mezzo del simbolo “`&`”. Il terzo campo:

```
<textarea name="comments" rows="10" cols="60">
</textarea>
```

non è concettualmente diverso dai primi due. Anche in questo caso sarà generata una coppia:

```
comments= <testo relativo ai commenti>
```

Gli attributi `rows` e `cols` regolano la dimensione della finestra di editing. Infine si definiscono i due bottoni di invio e di reset

```
<input type="submit" value="Invia Commenti">
<input type="reset" value="Cancella Tutto">
```

Si osserva che, a eccezione degli spazi, cui è riservato il simbolo speciale “+”, in virtù del fatto che gli spazi sono usati molto di frequente, i caratteri speciali sono rappresentati con la seguente sequenza, detta sequenza di escape:

`%`*`xx`*

dove *`xx`* è un numero di due cifre esadecimali che indica il codice ASCII del carattere speciale che si vuole rappresentare. Per esempio, il simbolo `%` stesso può essere inviato con la sequenza di escape:

```
&25
```

dove `25` nella codifica ASCII esadecimale corrisponde al simbolo “`%`”.

Alla fine ciò significa che il CGI che si vede recapitare i parametri con questo formato dovrà farsi carico di decifrare queste sequenze di escape.

16.7 Il CGI "Sono Graditi i Vostri Commenti"

Dei tanti modi di gestire per mezzo di un CGI il FORM discusso nell’esempio, riportiamo una implementazione dovuta a Thomas Boutell, l’autore della celebre libreria grafica GD per la costruzione di grafici e del programma freeware Mapedit. Il codice è riportato nel Listato 16.4 con molti commenti.

```
/* Indicare in quale directory inserire il file dei commenti */

#define COMMENT_FILE "c:\\http\\comments.txt"

#include <stdio.h>
#include <stdlib.h>

/* Variabili globali */

/* Numero massimo di campi gestiti nel form */
#define FIELDS_MAX 100

char *names[FIELDS_MAX];
char *values[FIELDS_MAX];

int fieldsTotal = 0;

/* Controlla che la richiesta provenga davvero da un modulo */
int VerifyForm();

/* Analizza i parametri passati, riempendo gli array names[]
e values[] con informazione utile                          */
void ParseForm();

/* Libera la memoria associata con i dati del form */
void FreeForm();

/* Copia src in dst trasformando le sequenze di escape
dei caratteri speciali                                */
void UnescapeString(char *dst, char *src);

int main(int argc, char *argv[])
{
  FILE *out;
  int i;
  int nameIndex = -1, emailIndex = -1, commentsIndex = -1;
```

```
  printf("Content-type: text/html\n\n");

  printf("<html>\n");
  printf("<head>\n");
  /* Controlla che sia un form inviato con metodo POST */
  if (!VerifyForm()) {
    printf("<title>Non è un form di tipo POST</title>\n");
    printf("</head>\n");
    printf("<h1>Non è un form di tipo POST</h1>\n");
    printf("</body></html>\n");
    return 0;
  }

  ParseForm();   /* OK, analizza il form. */

  /* Usa l'informazione */

  /* Trova l'indice di ogni campo nell'array */
  for (i = 0; (i < fieldsTotal); i++) {
    if (!strcmp(names[i], "name")) {
      nameIndex = i;
    } else if (!strcmp(names[i], "email")) {
        emailIndex = i;
    } else if (!strcmp(names[i], "comments")) {
        commentsIndex = i;
    }
  }

  /* Se manca un campo, segnalalo */
  if ((nameIndex == -1) || (emailIndex == -1) || (commentsIndex ==
-1)) {
    printf("<title></title>\n");
    printf("</head>\n");
    printf("<h1>Riempire tutti i campi</h1>\n");
    printf("Inserire nome, email e i commenti.\n");
    printf("Torna alla pagina precedente e riprova.\n");
    printf("</body></html>\n");
    return 0;
  }

/* OK, Scriviamo tutte le informazioni su un file
in modalità APPEND                                */

  out = fopen(COMMENT_FILE, "a");
  fprintf(out, "From: %s <%s>\n",
      values[nameIndex], values[emailIndex]);
  fprintf(out, "%s\n", values[commentsIndex]);

  printf("<title>Grazie, %s</title>\n", values[nameIndex]);
  printf("</head>\n");
  printf("<h1>Grazie, %s</h1>\n", values[nameIndex]);
  printf("Grazie per i Vostri Commenti.\n");
```

```
  printf("</body></html>\n");
  /* Libera la memoria usata */
  FreeForm();
  return 0;
}

int VerifyForm()
{
  char *contentType;
  char *requestMethod;
  int bad = 0;
  /* Verifica il content type dei dati ricevuti */
  contentType = getenv("CONTENT_TYPE");
  if  (strcmp(contentType,  "application/x-www-form-urlencoded")  !=
0) {
    bad = 1;
  }

  /* Controlla che si sia usato un metodo POST */
  requestMethod = getenv("REQUEST_METHOD");
  if (strcmp(requestMethod, "POST") != 0) {
    bad = 1;
  }

  return !bad;
}

/* Analizza l'informazione ricevuta e valorizza names e values[]*/

void ParseForm()
{
  char *contentLength = getenv("CONTENT_LENGTH");
  /* Numero di caratteri nei dati */
  int length;
  /* Prepara il buffer dove leggere i dati */
  char *buffer;
/* Posizione corrente nel buffer mentre si cercano i separatori */
  char *p;
  /* Determina la lunghezza dell'input */
  if (!contentLength) {
    length = 0;
  } else {
    length = atoi(contentLength);
    }
/* Alloca un buffer per memorizzare l'input. Include
uno spazio in più per semplificare l'analisi sintattica */
  buffer = (char *) malloc(length + 1);
  if (!buffer) {
    /* Uh-oh */
    return;
  }
  /* Legge tutti i dati da standard input */
```

```
  if (fread(buffer, 1, length, stdin) != length) {
    /* Uh-oh */
    return;
  }
  p = buffer;
  while (length) {

    char *name;   /* Inizio del nome corrente */
    char *value;  /* Inizio del valore corrente */

    int found;

    /* Si accerta che ci sia spazio per più campi */
    if (fieldsTotal == FIELDS_MAX) {
    /* Basta, non ne posso accettare altri */
      return;
    }

    name = p;

    /* Per prima cosa cerca il segno =. */
    found = 0;
    while (length) {
      if (*p == '=') {
       /* Termina il nome con un carattere null */
        *p = '\0';
        p++;
        found = 1;
        break;
      }
      p++;
      length--;
    }
    if (!found) {
    /* Un vuoto o una voce troncata. È strano ma potrebbe accadere
*/
      break;
    }
    value = p;
    /* Ora trova &. */
    found = 0;
    while (length) {
      if (*p == '&') {
       /* Termina il nome con un carattere null */
        *p = '\0';
        p++;
        found = 1;
        break;
      }
      p++;
      length--;
    }
```

```
    if (!found) {
      /* Suppone che sia la fine della coppia */
      *p = '\0';
    }

    names[fieldsTotal] = (char *) malloc(strlen(name) + 1);
    if (!names[fieldsTotal]) {
      /* Uh-oh, no memory.*/
      return;
    }
    values[fieldsTotal] = (char *) malloc(strlen(value) + 1);
    if (!values[fieldsTotal]) {
      /* Uh-oh, no memory. */
      free(names[fieldsTotal]);
      return;
    }

    /* Copia la stringa e risolve l'escape */
    UnescapeString(names[fieldsTotal], name);
    UnescapeString(values[fieldsTotal], value);
    fieldsTotal++;
    /* Continua con le altre coppie */
  }
  free(buffer); /* Libera il buffer. */
}

void FreeForm()
{
  int i;
  for (i=0; (i < fieldsTotal); i++) {
    free(names[i]);
    free(values[i]);
  }
}

void UnescapeString(char *dst, char *src)
{
/* Cicla sui caratteri della stringa finché non trova il null */
  while (*src) {
    char c;
    c = *src;
    /* Gestisce gli spazi rappresentati con + */
    if (c == '+') {
      c = ' ';
    } else if (c == '%') {
        /* Handle % escapes */
        char hexdigits[3];
        int ascii;
        src++;
        if (!*src) {
          /* Numeri mancanti, ignora l'escape */
          break;
```

```
        }
        hexdigits[0] = *src;
        src++;
        if (!*src) {
          /* Numeri mancanti, ignora l'escape */
          break;
        }
        hexdigits[1] = *src;
        /* Aggiunge un null finale... */
        hexdigits[2] = '\0';
        /* Usa la sscanf() per leggere il valore esadecimale */
        sscanf(hexdigits, "%x", &ascii);
        /* e lo converte nuovamente in carattere */
        c = ascii;
    }
    *dst = c;
    src++;
    dst++;
  }
  *dst = '\0';
}
```

Listato 16.4 Un CGI per gestire il FORM dei commenti

Per linee generali il programma segue il seguente flusso:

- • invia (come sempre) l’intestazione della pagina di ritorno;
- • verifica la correttezza del FORM che gli è stato inviato (VerifyForm). In questo caso specifico si assicura che sia stato scelto un metodo POST per l’invio dei parametri, altrimenti risponde con una pagina che segnala l’errore;
- • analizza i parametri del FORM (ParseForm). In pratica con questa funzione si valorizzano due array:

```
char *names[FIELDS_MAX];
char *values[FIELDS_MAX];
```

Ogni elemento del primo array punta a un nome di parametro secondo l’ordine di trasmissione e, corrispondentemente, ogni elemento del secondo punta al valore associato a quel nome di parametro. All’interno della funzione ParseForm viene invocata la funzione UnescapeString che provvede a decodificare le eventuali sequenze di escape trasmesse nella sequenza di coppie <nome=valore>;

- • infine, con la funzione FreeForm vengono liberati gli spazi di memoria allocati dinamicamente creati nel processo di acquisizione e interpretazione dei parametri.

16.8 Altri linguaggi di programmazione di Web Server

Il linguaggio C rimane il linguaggio di programmazione più diffuso e più usato nel mondo, nonostante l’avvento di altri linguaggi concorrenti come il C++ e più di recente Java. A onor del vero, però, è doveroso constatare che, almeno nello sviluppo di CGI, il linguaggio C di fatto non è in prima posizione quanto a diffusione e utilizzo: altri linguaggi si contendono questa piazza; tra di essi troviamo il Perl. Il Perl è un interprete in memoria molto flessibile e coinciso che consente di ridurre drasticamente il numero delle istruzioni di un CGI a parità di contenuto funzionale. Del resto i principali HTTP Server, e tra essi quello che ha oltre il 50% delle installazioni nel mondo, Apache, mettono a disposizione due principali interfacce di programmazione: una per il C e l’altra per il Perl.

Infine, due ulteriori osservazioni e una possibile conclusione.
In primo luogo, accanto all'approccio CGI tradizionale spesso se ne preferisce uno *servlet*, in cui cioè il Web Server colloquia con un server applicativo; il vantaggio principale è che in questo modo viene più semplicemente aperto un thread per connessione e non un processo per connessione, il che consente di rispondere più rapidamente alle richieste contemporanee di più client.
In secondo luogo, i Web Server sono sempre più spesso chiamati a ricercare informazioni strutturate – numeri di telefono e indirizzi, dati scientifici, amministrativi, finanziari ecc. – e dunque hanno a che fare con grandi basi di dati già esistenti, cui si ha accesso attraverso il linguaggio SQL; quindi i programmi CGI o servlet dovranno invocare comandi SQL per interagire con tali database. In entrambi i casi, attualmente la via meno legata alle singole case produttrici e quindi più standard sembra quella di realizzare CGI e servlet con il linguaggio Java,, accedendo alle basi di dati, per esempio, tramite JDBC (*Java DataBase Connection*).
Ma mentre il mondo della Rete è in continua rapida evoluzione, il C prosegue la sua corsa spigliata dopo essersi levato la soddisfazione di costituire la matrice indelebile da cui sono nati molti "nuovi" linguaggi.

16.9 Esercizi

1. 1. Scrivere un CGI che accetti in ingresso la data di nascita e calcoli il numero di anni e giorni trascorsi da quella data a oggi.

2. 2. Realizzare un CGI che legga lo stato attuale da un file di log e lo presenti in forma di pagine HTML.

3. 3. Scrivere tramite CGI un sistema che consenta:
- • la registrazione
- • la cancellazione
- • la correzione

di un utente che intende abbonarsi a un generico servizio. [*Suggerimento*: Usare file tradizionali.]

4. 4. Scrivere un CGI che, basandosi su un metodo di generazione di numeri casuali, produca ogni volta una pagina con un colore diverso.

Soluzione degli esercizi

Capitolo 1

5
```
#include <stdio.h>
#include <math.h>

#define ZERO 0
#define TOP  1000

main()
{
  int a, b, c, x, y;
```

```
  printf("Valore di x: ");
  scanf("%d", &x);

  printf("Valore di y: ");
  scanf("%d", &y);

  a = ZERO-abs(x);
  b = TOP-abs(y);
  c = a*b;

  printf("Valore di a: %d\n", a);
  printf("Valore di b: %d\n", b);
  printf("Valore di c: %d\n", c);
}
```

Capitolo 2

2

```
/* Determina il maggiore tra quattro valori */
#include <stdio.h>
main()
{
int a, b, c, d;
printf("\nDigita quattro valori interi distinti: ");
scanf("%d", &a);
scanf("%d", &b);
scanf("%d", &c);
scanf("%d", &d);

if(a>b)
  if(a>c)
    if(a>d)
      printf("Il maggiore è: %d\n", a);
    else
      printf("Il maggiore è: %d\n", d);
  else
    if(c>d)
      printf("Il maggiore è: %d\n", c);
    else
      printf("Il maggiore è: %d\n", d);
else
  if(b>c)
    if(b>d)
      printf("Il maggiore e': %d\n", b);
    else
      printf("Il maggiore è: %d\n", d);
  else
    if(c>d)
      printf("Il maggiore è: %d\n", c);
    else
      printf("Il maggiore è: %d\n", d);
}
```

4

```
/* Determina il maggiore e il minore tra tre valori */
```

```c
#include <stdio.h>
main()
{
int a, b, c;
printf("\nDigita tre valori interi distinti: ");
scanf("%d", &a);
scanf("%d", &b);
scanf("%d", &c);

if(a>b)
  if(a>c) {
    printf("Il maggiore è: %d\n", a);
    if(b<c)
      printf("Il minore è: %d\n", b);
    else
      printf("Il minore è: %d\n", c);
  }
  else {
    printf("Il maggiore è: %d\n", c);
    printf("Il minore è: %d\n", b);
  }
else
  if(b>c) {
    printf("Il maggiore è: %d\n", b);
    if(a<c)
      printf("Il minore è: %d\n", a);
    else
      printf("Il minore è: %d\n", c);
  }
  else {
    printf("Il maggiore è: %d\n", c);
    printf("Il minore è: %d\n", a);
  }
}
```

5

```c
/* Verifica il valore delle espressioni date */
#include <stdio.h>
int a, b, c, ris;
main()
{
a = 5;
b = 35;
c = 7;

ris = a+b*c;
printf("\n a) %d", ris);

ris = a>b;
printf("\n b) %d", ris);

ris = (a+b) * (a<b);
printf("\n c) %d", ris);

ris = (a+b) && (a<b);
printf("\n d) %d", ris);

ris = (a+b) || (a>b);
printf("\n e) %d", ris);
```

```
ris = ((a*c) - b) || (a>b);
printf("\n f) %d", ris);

ris = ((a*c) != b) || (a>b);
printf("\n g) %d", ris);

ris = (a>b) || (a<c) || (c==b);
printf("\n h) %d\n", ris);

/* O ALTERNATIVAMENTE ALL"INTERNO DELLE PRINTF */
printf("\n a) %d", a+b*c);
printf("\n b) %d", a>b);
printf("\n c) %d", (a+b) * (a<b));
printf("\n d) %d", (a+b) && (a<b));
printf("\n e) %d",  (a+b) || (a>b));
printf("\n f) %d", ((a*c) - b) || (a>b));
printf("\n g) %d", ((a*c) != b) || (a>b));
printf("\n h) %d\n", (a>b) || (a<c) || (c==b));
}
```

I risultati visualizzati dal programma saranno i seguenti:

```
a)    250
     b)    0
     c)    40
     d)    1
     e)    1
     f)    0
     g)    0
     h)    1
```

6

I risultati visualizzati dal programma saranno i seguenti:

```
a) Vero (1)
     b) Falso (0)
     c) Vero (1)
     d) Vero (1)
     e) Falso (0)
     f) Vero (1)
```

7

I risultati visualizzati dal programma saranno:

```
a)    vero (1)
     b)    falso (0)
     c)    vero (1)
     d)    vero (1)
     e)    vero (1)
     f)    falso (0)
```

8

```
d = (a*2+b) != c ? a*b*c: a+b+c;
```

9

```
/* Esempio visualizzazione menu e controllo scelte utente */
#include <stdio.h>
main()
{
```

```
char scelta;
printf("\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n");
printf("\n              MENU DI PROVA\n");
printf("\n          a) Per immettere dati");
printf("\n          b) Per determinare il maggiore");
printf("\n          c) Per determinare il minore");
printf("\n          d) Per ordinare");
printf("\n          e) Per visualizzare");
printf("\n\n                              Scelta: ");
scelta = getchar();

switch(scelta) {
  case 'a':
    printf("\n In esecuzione l'opzione a");
    break;
  case 'b':
    printf("\n In esecuzione l'opzione b");
    break;
  case 'c':
    printf("\n In esecuzione l'opzione c");
    break;
  case 'd':
    printf("\n In esecuzione l'opzione d");
    break;
  case 'e':
    printf("\n In esecuzione l'opzione e");
    break;
  default:
    printf("\n Opzione inesistente");
    break;
}
}
```

10

```
printf("\n Il maggiore è: %d", (x=(a>b)?a:b)?c:x:c);
```

Dove x è una variabile `int`.

Capitolo 3

1

```
/* Determina il fattoriale dei numeri minori uguali
   all'intero immesso dall'utente                  */
#include <stdio.h>
main()
{
int n, fat, aux;

printf("CALCOLO DEI FATTORIALI DEI NUMERI <= N\n\n");
printf("Inser. n: ");
scanf("%d", &n);

fat = 1;
printf("Il fattoriale di: 0 ha valore: %d\n", fat);
```

```
for(aux=1; aux<=n; aux++) {
  fat = fat*aux;
  printf("Il fattoriale di: %d ha valore: %d\n", aux, fat);
}
}
```

2

```
/* Determina il maggiore, il minore e la media
   dei valori immessi                            */
#include <stdio.h>
#include <limits.h>
main()
{
int i, n, numero, max, min, media;

printf("MAGGIORE MINORE E MEDIA\n");
min = INT_MAX;
max = INT_MIN;
media = 0;
i = 1;

do {
  printf("\nLunghezza della sequenza: ");
  scanf("%d", &n);
}
while(n<1);

for(i=1; i<=n; i++) {
  printf("Valore int.: \t");
  scanf("%d", &numero);
  if(numero>max)
    max = numero;
  else
    if(numero<min)
      min = numero;
  media = media+numero;
}

printf("Maggiore: %d\n", max);
printf("Minore: %d\n", min);
printf("Media: %d\n", media/n);
}
```

3

```
/* Visualizza un rettangolo di cornice * e
   parte interna Q, le dimensioni del
   rettangolo sono decide dall'utente     */
#include <stdio.h>
main()
{
int i, j, linee, colonne;

do {
  printf("\nNumero di linee: ");
  scanf("%d", &linee);
}
while(linee<1);

do {
  printf("\nNumero di colonne: ");
```

```
  scanf("%d", &colonne);
}
while(colonne<1);

for(i=1; i<=linee; i++)
  for(j=1; j<=colonne; j++)
    if(i==1 || i==linee || j==1 || j==colonne) {
      printf("*");
      if(j==colonne) printf("\n");
    }
    else  printf("Q");
}
```

4

```
/* Visualizza tanti rettangoli quanti ne
   desidera l'utente con caratteri e dimensioni
   scelti a tempo di esecuzione                */
#include <stdio.h>
main()
{
int i, j, y, linee, colonne, volte;
char cornice, interno;

do {
  printf("\nNumero di linee: ");
  scanf("%d", &linee);
}
while(linee<1);

do {
  printf("\nNumero di colonne: ");
  scanf("%d", &colonne);
}
while(colonne<1);

printf("\nCarattere della cornice: ");
scanf("%1s", &cornice);
printf("\nCarattere dell'interno: ");
scanf("%1s", &interno);

do {
  printf("\nNumero di visualizzazioni: ");
  scanf("%d", &volte);
}
while(colonne<1);

for(y=1; y<=volte; y++)
  for(i=1; i<=linee; i++)
    for(j=1; j<=colonne; j++)
      if(i==1 || i==linee || j==1 || j==colonne ) {
        printf("%c", cornice);
        if(j==colonne) printf("\n");
      }
      else printf("%c", interno);
}
```

Capitolo 4

1.

Il ciclo che effettua la somma deve essere realizzato in modo che l'elemento del secondo array sia simmetrico rispetto al primo.

```
for(i=0; i<n; i++)
       c[i] = a[i] + b[n-i-1];
```

2.

```
max = voti[0];
     min = voti[0];
     media = voti[0];
     for(i = 0;i <= 5; i++) {
       if(voti[i]>max)
         max = voti[i];
       else
         if(voti[i]<min)
           min = voti[i];
      media = media+voti[i];
     }
```

13.

Devono essere definite le dimensioni della matrice.

```
#define N 10
     #define P 10
     #define M 10

     int mat1[N][P];
     int mat2[P][M];
     int pmat[N][M];
```

Si devono richiede all'utente le reali dimensioni e si deve controllare che il loro valore non superi le dimensione delle matrici. I valori da richiede sono soltanto tre in quanto le colonne della prima matrice devono essere in numero uguale alle righe della seconda.

```
/* Richiesta delle dimensioni */
     do {
       printf("Numero di linee I matrice: ");
       scanf("%d", &n);
     }
     while((n>=N) || (n<1));

     do {
       printf("Numero colonne I matrice / righe II matrice: ");
       scanf("%d", &p);
     }
     while((p>=P) || (p<1));

     do {
       printf("Numero di colonne II matrice: ");
       scanf("%d", &m);
     }
     while((m>=M) || (m<1));
```

Anteriormente devono essere state dichiarate le variabili `n`, `m` e `p`.

```
int n, m, p;
```

Sostituire `N`, `M` e `P` con `n`, `m` e `p` nel resto del programma.

16.

```
/* Calcolo media voti per studente e per prova
   Nell'esemplificazione utilizziamo 3 studenti e 4 prove */
#include <stdio.h>

#define n 4
#define m 5
float voti[n][m];

main()
{
int i, j;

printf("\n \n CARICAMENTO DEI VOTI \n \n");
for(i=0; i<n-1; i++)
  for(j=0; j<m-1; j++) {
    printf("Ins. studente %d prova %d: ", i+1, j+1);
    scanf("%f", &voti[i][j]);
  };

/* Calcolo medie per studente */
for(i=0; i<n-1; i++) {
  voti[i][m-1] = 0;
  for(j = 0; j < m-1; j++)
    voti[i][m-1] = voti[i][m-1] + voti[i][j];
  voti[i][m-1] = voti[i][m-1] / (m-1);
}

/* Calcolo medie per prova */
for(j=0; j<m; j++) {
  voti[n-1][j] = 0;
  for(i=0; i<n-1; i++)
    voti[n-1][j] = voti[n-1][j] + voti[i][j];
  voti[n-1][j] = voti[n-1][j]/(n-1);
}

printf("\n \n VISUALIZZAZIONE DELLA MATRICE \n ");
for(i=0; i<n; i++) {
  printf("\n");
  for(j=0; j<m; j++)
    printf("%8.3f", voti[i][j]);
}
putchar('\n'); putchar('\n');
}
```

Esempio di esecuzione

```
Ins. studente 1 prova 1: 4
     Ins. studente 1 prova 2: 5
     Ins. studente 1 prova 3: 4
     Ins. studente 1 prova 4: 7
     Ins. studente 2 prova 1: 8
     Ins. studente 2 prova 2: 10
     Ins. studente 2 prova 3: 8
     Ins. studente 2 prova 4: 10
```

```
Ins. studente 3 prova 1: 6
Ins. studente 3 prova 2: 7
Ins. studente 3 prova 3: 8
Ins. studente 3 prova 4: 6


 VISUALIZZAZIONE DELLA MATRICE

   4.000   5.000   4.000   7.000   5.000
   8.000  10.000   8.000  10.000   9.000
   6.000   7.000   8.000   6.000   6.750
   6.000   7.333   6.667   7.667   6.917
```

Capitolo 5

1.

Qualsiasi soluzione si adotti tra quelle proposte nel testo, l'operatore relazionale dell'`if`, che controlla lo scambio di valori tra gli elementi deve essere cambiato da `>` a `<`.

```
if(vet[i]<vet[i+1])
```

Non ci sono altre modifiche da effettuare.

2.

```
/* Ricerca di un valore in una matrice */
#include <stdio.h>

#define N 10
#define M 10
char alfa[N][M];

main()
{
int n, m, i, j, k;
char ric;

/* Richiesta delle dimensioni */
do {
  printf("Numero di linee: ");
  scanf("%d", &n);
}
while((n>=N) || (n<1));

do {
  printf("Numero di colonne: ");
  scanf("%d", &m);
}
while((m>=M) || (m<1));

printf("\n \n CARICAMENTO DELLA MATRICE \n \n");
for(i=0; i<n; i++)
  for(j=0; j<m; j++) {
    printf("Ins.carattere nella linea %d colonna %d val:", i, j);
    scanf("%1s", &alfa[i][j]);
```

```
};

/* Richiesta del carattere da ricercare */
printf("\n \n Carattere da ricercare: ");
scanf("%1s", &ric);

printf("\n \n VISUALIZZAZIONE DELLA MATRICE \n ");
for(i=0; i<n; i++) {
  printf("\n");
  for(j = 0; j < m; j++)
    printf("%3c", alfa[i][j]);
}
printf("\n\n");

/* Ricerca del carattere all'interno della matrice */
k = 0;
for(i=0; i<n; i++)
  for(j=0; j<m; j++) {
    if(alfa[i][j]==ric) {
      printf("%c in linea %d colonna %d\n", ric, i+1, j+1);
      k = 1;
    }
  };
if(k==0)  printf("%c non presente nella matrice", ric);
}
```

Esempio di esecuzione

```
 Carattere da ricercare: a


      VISUALIZZAZIONE DELLA MATRICE

       t  b  a
       m  d  g
       a  k  k
       d  a  m
       v  f  g

     a in linea 1 colonna 3
     a in linea 3 colonna 1
     a in linea 4 colonna 2
```

Capitolo 6

3

```
/* Concatenazione di dei primi n caratteri di una stringa
   su di una altra  con strcat */
#include <stdio.h>
#include <string.h>

char frase[160] = "Analisi, requisiti";

main()
{
char dimmi[80];
```

```
int i;

for(i=0; ((dimmi[i]=getchar())!='\n') && (i<80); i++)
  ;
dimmi[i] = '\0';
strncat(frase, dimmi, 5);
printf("%s \n", frase);
}
```

4

```
/* Confronto dei primi n caratteri di due stringhe con strncmp */
#include <stdio.h>
#include <string.h>

char prima[160] = "Analisi, requisiti";

main()
{
char seconda[80];
int i, x;

for(i=0; ((seconda[i]=getchar())!='\n') && (i<80); i++)
  ;
seconda[i]='\0';
if((x=(strncmp(prima, seconda, 5)))==0)
  printf("Sono uguali\n");
else
  if(x>0)
    printf("la prima è maggiore della seconda\n");
  else
    printf("la seconda è maggiore della prima\n");
}
```

Capitolo 7

1

```
double pot(double base, int esp)
{
double po;

po = 1;
if( esp == 0 )
  return( 1 );
else {
  while( esp-- ) po *= base;
  return( po );
}
}
```

2

```
#include <stdio.h>

void message (int, int);
```

```
char messaggio[] = "Ciao, baby";

main()
{
  int riga = 10;
  int colonna = 20;
  message(riga, colonna);
}

void message(int r, int c)
{
  while( c-- )  printf("\n");
  while( r-- )  putchar(' ');
  printf("%s\n\n", messaggio);
}
```

3

```
#include <stdio.h>
char buf[128];
void min_man(void);

main()
{
  printf("\nInserisci stringa: ");
  scanf("%s", buf);

  min_man();
  printf("%s\n", buf);
 }

void min_man(void)
{
  int i;

  for ( i = 0; buf[i] != '\0'; i++)
    if (buf[i] >= 'a' && buf[i] <= 'z')
      buf[i] = buf[i] - 'a' + 'A';
}
```

4

```
#include <stdio.h>
int numeri = 0;
int alfa   = 0;
char buf[80];
int i;

void num_alfa(void);

main()
{
  printf("\nInserisci stringa: ");
```

```
  scanf("%s", buf);

  num_alfa();
  printf("Ci sono %2d caratteri numerici\n", numeri);
  printf("Ci sono %2d caratteri alfabetici\n", alfa);
}

void num_alfa(void)
{
  for (i = 0; buf[i] != '\0'; i++)
    switch(buf[i]) {
      case '0':
      case '1':
      case '2':
      case '3':
      case '4':
      case '5':
      case '6':
      case '7':
      case '8':
      case '9':
                 numeri++;
                 break;
      default:
                 alfa++;
                 break;
    }
}
```

5

È sufficiente aggiungere alla fine della funzione `immissione` la chiamata alla procedura, già presente nel programma, che effettua l'ordinamento, passandole il numero di elementi che la compongono.

```
int immissione2()
{
...
ordinamento( n );
return( n );
}
```

Capitolo 9

1

```
#include <stdio.h>
int a[5] = { 1, 2, 3, 4, 5 };

main()
{
  int i, *p;
  p = a;
  printf("Gli elementi del vettore sono:\n\n");
```

```
  for (i = 0; i <= 4; i++)
    printf("a[%d] = %d\n", i, *p++);
}
```

2

```
#include <stdio.h>

char *vet[] = {"Messaggio #1\n",
               "Messaggio #2\n",
               "Messaggio #3\n",
               "Messaggio #4\n",
               "Messaggio #5\n",
               NULL
              };
char **p = vet;

main()
{
  while(*p != NULL)
    printf("%s", *p++);
}
```

3

```
char *str_in_str(char *s, char *t)
{
  char *v;

  while(*s != '\0') {
    if(*s == *t)
      for(v = t; *s == *v;)  {
        if(*++v == '\0') return(s-(v-t)+1);
        if(*s++ == '\0') return(NULL);
      }
    else
      s++;
  }
  return(NULL);
}
```

4

```
/* versione 1  che usa gli array*/
strcop( char s[], char t[])
{
  int i=0;
  while ((s[i] = t[i]) != '\0') i++;
}

/*versione 1.1 che usa i puntatori*/
strcop(char *s, char *t)
{
  while ((*s = *t) != '\0') {s++; t++;}
}
```

```
/* versione 2 */
strcop(char *s, char *t)
{
  while ((*s++ = *t++) != '\0') ;
}

/* versione 3 */
strcop(char *s, char *t)
{
  while (*s++ = *t++);
}
```

Ovviamente la migliore delle versioni è la terza, anche se abbastanza criptica.

5

```
#include <stdio.h>
#include <malloc.h>

main()
{
  char *s;
  int  n;
  printf("Inserire dimensione del buffer : ");
  scanf("%d", &n);
  s = (char*) malloc(n+1);
  ...
}
```

6

La funzione viene così dichiarata:

```
void immissione( int *, int * );
```

Il primo parametro è il puntatore alla variabile `n` di `gestione_sequenza` dove immissione memorizzerà, mediante il puntatore `pn`, la lunghezza della sequenza.

```
void immissione( int *pn, int *vet )
{
int i, n;
char invio;

do {
  printf("\nNumero elementi: ");
  scanf("%d", &n);
}
while (n < 1 || n > MAX_ELE);

for(i = 0; i < n; i++) {
  printf("\nImmettere un intero n.%d: ",i);
  scanf("%d", &vet[i]);
}

*pn = n;
}
```

Capitolo 10

3

```
/* Calcolo della potenza di base elevato a esponente
   dove esponente è un numero maggiore uguale a zero */
#include <stdio.h>

double potenza(float, int);

main()
{
float b; int e;
printf("\n\n Calcolo della potenza \n\n");
printf("Inser. base: \t");
scanf("%f", &b);
printf("Inser. esponente: \t");
scanf("%d", &e);
printf("Potenza: %lf\n", potenza(b, e));
}

/* Funzione per il calcolo di base elevato a esp con esp>=0 */
double potenza(float base, int esp)
{
double pot = 1;
if(esp==0)  return(1);   /* caso esponente uguale a zero */
if(esp>0) {              /* calcolo della potenza */
  do
    pot = pot*base;      /* base*base*base.... esp volte */
  while(--esp>0);
}
return(pot);
}
```

5

```
/* Funzione il calcolo di base elevato a esp
   esp può essere un intero qualsiasi             */
double potenza(float base, int esp)
{
int s = 1;
double pot = 1;        /* inizializzazione di pot a 1
                          caso di esponente uguale a zero */
if(esp<0) {            /* l'esponente è negativo ? */
  esp = abs(esp);
  s = 0;
}

if(esp>0) {            /* calcolo della potenza */
  do
    pot = pot*base;    /* b*b*b... n volte */
  while(--esp>0);
}

if(s) return(pot);     /* l'esponente è positivo ? */
else  return(1/pot);
}
```

Per utilizzare la funzione `abs` si deve aver incluse la libreria `math.h`.

6

```
/* Funzione ricorsiva per il calcolo di base elevato a esp
   esp può essere un intero qualsiasi                       */
double potenza(float base, int esp)
{
if(esp==0)
  return(1);
else
  if(esp<0)
    return(1/potenza(base, -esp));
  else
    return(base*potenza(base, esp-1));
}
```

7

```
/* Massimo comun denominatore di due interi positivi */
#include <stdio.h>
int mcd(int, int);

main()
{
int t, k;
printf("\n\n Calcolo del massimo comun divisore \n\n");
printf("Inser. t: \t");
scanf("%d", &t);
printf("Inser. k: \t");
scanf("%d", &k);
printf("Massimo comun divisore: %d\n", mcd(t, k));
}

/* Funzione ricorsiva per il calcolo del massimo comun divisore */
int mcd(int t, int k)  {
if(k==0)
  return(t);
else
  if(k>t)
    return(mcd(k, t));
  else
    return(mcd(k, t%k));
}
```

Capitolo 13

1

```
#include <stdio.h>
main()
{
FILE *fp;
char buffer[81];

fp = fopen("c:\\autoexec.bat", "r");
printf("\nIl file autoexec.bat contiene: ");
```

```
while(fgets(buffer, 80, fp) !=NULL) printf(buffer);
fclose(fp);
}
```

2

```
#include <stdio.h>
char buffer[80] = "prova d'uso di fwrite\n Questa è la seconda
riga.\n";
main()
{
int numwrite;
FILE *fp;
char nomefile[80];

printf("Digitare il nome del file: ");
gets(nomefile);
fp = fopen(nomefile, "w");
numwrite = fwrite(buffer, sizeof(char), 80, fp);
printf("%d caratteri scritti sul file %s\n", numwrite, nomefile);
printf("Controllare visualizzando il file con un editor\n");
printf("del sistema operativo in uso (esempio type, vi o
wordpad)\n");
}
```

3

Come in `Procedura Anagrafica`, basta sostituire la struttura

```
struct per
```

con le strutture

```
struct dat
{int giorno, mese, anno};

typedef struct dat Data;
struct student {
  char *cog-stud;
  char *nom_stud;
  char *ind_stud;
  Data dat_nasc;
};
```

4

```
#include <stdio.h>
#include <stdlib.h>
main()
{
FILE *fp;
char nomefile[80], buffer[81];

printf("Digitare il nome del file:");
gets(nomefile);
```

```
/* Apre il file */
if((fp = fopen(nomefile, "r")) == NULL){
  perror("Mancata apertura del file");
  exit(1) ;
}
/* legge e visualizza una riga */
fgets(buffer, 80, fp);
printf("La linea letta e' : %s", buffer);

/* ora ritorna all'inizio e rilegge una riga*/
fseek(fp, 0L, 0);
fgets(buffer, 80, fp);
printf("La linea riletta e' : %s", buffer);
}
```

5

```
#include <stdio.h>
main()
{
int i;
FILE *fp;
char sequenza[80];

/* Apre   il file.  Il simbolo di root \ si ottiene   con il
doppio slash */
fp = fopen("c:\\autoexec.bat", "r");
printf("Le prime 10 sequenze di caratteri sono:\n");
for(i = 0; i < 10; i++) {
  if (fscanf(fp, "%s", sequenza) == EOF){
    printf("Fine file!\n");
    break;
  }
  else
    printf("Sequenza %d = \"%s\"\n", i, sequenza);
}
}
```

EOF è una costante di libreria che rappresenta la End Of File, cioè la fine del file.

Capitolo 14

1

Presentiamo due possibili soluzioni che si differenziano per il modo con cui i risultati vengono comunicati dalla funzione al chiamante. Nella prima versione (`conta_pari`) i due parametri vengono passati per indirizzo; nella seconda (`conta_pari2`) la funzione è di tipo `int` e ritorna il numero di pari, mentre il numero dei dispari è ancora restituito per indirizzo.

```
struct elemento {
  int inf;
  struct elemento *pun;
};

struct elemento *crea_lista2();
void visualizza_lista(struct elemento *);
void conta_pari(struct elemento *, int *, int *);
int conta_pari2(struct elemento *, int *);

main()
{
int pari, dispari;
struct elemento *punt_lista;

punt_lista = crea_lista2();
visualizza_lista(punt_lista);

/* chiamata prima versione di conta pari */
conta_pari(punt_lista, &pari, &dispari);
printf("\nPari: %d   Dispari: %d", pari, dispari);

/* chiamate seconda versione di conta pari */
printf("\nPari: %d", conta_pari2(punt_lista, &dispari));
printf("   Dispari: %d\n", dispari);
}

void conta_pari(struct elemento *p, int *ppari, int *pdispari)
{
*ppari = *pdispari = 0;

while(p!=NULL) {
  if(p->inf % 2 == 0)
    (*ppari)++;
  else
    (*pdispari)++;
  p = p->pun;
}
}

conta_pari2(struct elemento *p, int *pdispari)
{
int pari  = 0;
*pdispari = 0;

while(p!=NULL) {
  if(p->inf % 2 ==0)
    pari++;
  else
    (*pdispari)++;
  p = p->pun;
}
return(pari);
}
```

Si noti come la funzione `elimina_pari` modifichi il valore al puntatore iniziale della lista nel caso che il primo elemento contenga un numero pari; per questa ragione è necessario utilizzare un puntatore ausiliario paus per scorrere in avanti la lista e restituire comunque al chiamante il corretto riferimento all'inizio della lista.

```
#include <stdio.h>
#include <malloc.h>

struct elemento {
  int inf;
  struct elemento *pun;
};

struct elemento *crea_lista2();
void visualizza_lista(struct elemento *);
struct elemento *elimina_pari(struct elemento *);

main()
{
struct elemento *punt_lista;

punt_lista = crea_lista2();
visualizza_lista(punt_lista);

punt_lista =  elimina_pari(punt_lista);
visualizza_lista(punt_lista);
}

struct elemento *elimina_pari(struct elemento *p)
{
struct elemento *paus;
int logica = 1;

while(p!=NULL && logica)
  if(p->inf % 2 == 0)
    p = p->pun;
  else
    logica = 0;

paus = p;
while(paus->pun != NULL)
  if(paus->pun->inf % 2 == 0)
    paus->pun = paus->pun->pun;
  else
    paus = paus->pun;
return(p);

  }
```

4

```
#include <stdio.h>
#include <malloc.h>

struct elemento {
  int inf;
  struct elemento *pun;
};

void visualizza_lista(struct elemento *);
struct elemento *aggiungi(struct elemento *, struct elemento);
```

```
main()
{
struct elemento *punt_positivi = NULL;
struct elemento *punt_negativi = NULL;
struct elemento x;

do {
  printf("\nInserisci un informazione (0 per fine lista): ");
  scanf("%d", &x.inf);

  if(x.inf>0)
    punt_positivi = aggiungi(punt_positivi, x);
    if(x.inf<0)
      punt_negativi = aggiungi(punt_negativi, x);
}
while(x.inf!=0);

visualizza_lista(punt_positivi);
visualizza_lista(punt_negativi);
}


struct elemento *aggiungi(struct elemento *p, struct elemento x)
{
struct elemento *paus;

if(p==NULL) {
  p = (struct elemento *)malloc(sizeof(struct elemento));
  p->inf = x.inf;
  p->pun = NULL;
}
else {
  paus = (struct elemento *)malloc(sizeof(struct elemento));
  paus->inf = x.inf;
  paus->pun = p;
  p = paus;
}
return(p);
}
```

Capitolo 15

1

```
void differito(struct nodo *p)
{
if(p!=NULL) {
  differito( p->alb_sin );
  differito( p->alb_des );
  printf("%d  ", p->inf);
}
}
```

2

```
struct nodo *crea_nodo2(struct nodo *p, int val)
{

if(p==NULL) {
  p = (struct nodo *) malloc( sizeof( struct nodo ) );
  p->inf = val;
  p->occorrenze = 1;
  p->alb_sin = NULL;
  p->alb_des = NULL;
}
else {
  if(val > p->inf)
    p->alb_des = crea_nodo2(p->alb_des, val);
  else
    if(val < p->inf)
      p->alb_sin = crea_nodo2(p->alb_sin, val);
    else
      ++p->occorrenze;
}
return(p);
}
```

3

```
void simmetrico(struct nodo *p)
{
if(p!=NULL) {
  simmetrico( p->alb_sin );
  printf("\n%d  %d", p->inf, p->occorrenze);
  simmetrico( p->alb_des );
}
   }
```

4

```
void differito(struct nodo *p)
{
struct nodo *paus = p;
paus = paus->p_arco;

printf("(");
while(paus!=NULL) {
  differito(paus->figlio);
  paus = paus->p_arco;
}
printf("%c)", p->inf);
}
```

7

Per evitare di cadere in un ciclo infinito, ogni nodo del percorso deve essere visitato una e una sola volta. A questo scopo si usa un array (`gia_visitato[]`) che contiene tanti elementi interi quanti sono i nodi del grafo memorizzati nell'array `s[]` e nello stesso ordine di `s[]`. Tutto l'array viene inizializzato a zero, prima di transitare per un certo nodo ci si domanda se l'elemento corrispondente di `gia_visitato` ha valore uno, se così non è si assegna uno a quell'elemento e lo si visita.
Devono essere dichiarate le seguenti funzioni e variabili globali:

```
void giro( char );
void vai( int );

int gia_visitati[10] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

main()
{
char etichetta, invio;

mat_adiacenze();
vis_mat_adiacenze();
successori();
visita();

/* VISITA A PARTIRE DA UN NODO */
printf("\nImmettere l'etichetta di un nodo: ");
scanf("%c", &invio);
scanf("%c", &etichetta);
giro( etichetta );
}

void giro(char e)
{
int i = 0;
printf("\n");
while(i<n && s[i].inf!=e) i++;
if (i==n) return;
vai(i);
}


void vai(int i)
{
struct successore *p;

p = s[i].pun;
while(p!=NULL) {
  if(gia_visitati[p->inf]==0) {
    gia_visitati[p->inf] = 1;
    printf(" %c", s[p->inf].inf);
    vai( p->inf );
  }
  p = p->pun;
}
}
```

Appendice B: Parole chiave del C

auto

```
break
case
char
const
continue
default
do
double
else
enum
extern
float
for
goto
if
int
long
register
return
short
signed
sizeof
static
struct
switch
typedef
union
unsigned
void
volatile
while
```

Appendice C: File header

Ogni funzione è associata ad uno o più *file header* che devono essere inclusi ogni volta che si fa uso di quella funzione. Tali file contengono: le dichiarazioni delle funzioni correlate, delle macro, dei tipi dati e la definizione di costanti necessari all'esecuzione di un insieme di funzioni di libreria. La definizione di queste ultime dipende poi dall'implementazione del compilatore in uso. Di seguito riportiamo l'elenco dei file header standard, di molti dei quali abbiamo parlato estesamente nel testo.

File header	**Area di riferimento**
`<assert.h>`	Diagnostica.
`<ctype.h>`	Controllo e conversione caratteri. Per esempio: `isdigit(c)` ritorna un valore diverso da zero se `c` è una cifra decimale; analogamente operano `isalpha(c)`, `isspace(c)`, `isupper(c)` ecc.
`<Errno.h>`	Segnalazioni di errore.
`<float.h>`	Floating point.
`<limists.h>`	Definisce alcune costanti come `INT_MAX` e `INT_MIN` che contengono rispettivamente il massimo e il minimo valore rappresentabile con un `int`.
`<locate.h>`	Inizializzazione dei parametri locali.

<math.h>	Funzioni matematiche in doppia precisione.
<setjmp.h>	Salti non locali. Contiene la dichiarazione di funzioni che permettono di alterare l'esecuzione della normale sequenza di chiamata e uscita di una funzione, per esempio per obbligare a un ritorno immediato da una chiamata di funzione profondamente annidata.
<signal.h>	Gestione segnali. Contiene la dichiarazione di funzioni per la gestione di condizioni di eccezione che si verificano durante l'esecuzione, come l'arrivo di un segnale di interrupt da una sorgente esterna, oppure un errore nell'esecuzione.
<stdarg.h>	Gestione lista di argomenti variabili in numero e tipo. Contiene funzioni e/o macro che permettono di scandire tali liste, quindi può essere utile a sua volta per la realizzazione di funzioni che accettano un numero variabile di parametri.
<stddef.h>	Definizioni standard. Per esempio contiene la definizione di `ptrdiff_t` in grado di contenere la differenza, con segno, tra due puntatori e `size_t` il tipo (intero privo di segno) prodotto dalla funzione `sizeof`.
<stdio.h>	Input e Output. Funzioni quali `printf` e `scanf`.
<stdlib.h>	Utilità generale. Per esempio le funzioni per la conversione dei numeri, come `atof`, che trasforma una stringa in un `double`, o `rand` che ritorna un numero pseudo causale.
<string.h>	Gestione di stringhe. Funzioni quali `strcpy`, che consente di copiare una stringa su un altra e `strcat` che concatena due stringhe.
<time.h>	Gestione della data e dell'ora. Per esempio la funzione `time` che ritorna l'ora corrente.

Lo standard ANSI garantisce che inclusioni multiple di uno stesso file header standard non portano alcun effetto negativo e che l'ordine di inclusione è ininfluente. Dunque un header può essere incluso in qualsiasi ordine e un qualsiasi numero di volte; come sappiamo, deve comunque essere incluso prima che venga utilizzata una qualsiasi entità in esso definita.

Codice ASCII

0		32		64	@	96	`	128	Ç	160	á	192	└	224	Ó
1	☺	33	!	65	A	97	a	129	ü	161	í	193	┴	225	ß
2	☻	34	"	66	B	98	b	130	é	162	ó	194	┬	226	Ô
3	♥	35	#	67	C	99	c	131	â	163	ú	195	├	227	Ò
4	♦	36	$	68	D	100	d	132	ä	164	ñ	196	─	228	õ
5	♣	37	%	69	E	101	e	133	à	165	Ñ	197	┼	229	Õ
6	♠	38	&	70	F	102	f	134	å	166	ª	198	ã	230	µ
7	•	39	'	71	G	103	g	135	ç	167	º	199	Ã	231	þ
8	◘	40	(	72	H	104	h	136	ê	168	¿	200	╚	232	Þ
9	○	41	)	73	I	105	i	137	ë	169	®	201	╔	233	Ú
10	◙	42	*	74	J	106	j	138	è	170	¬	202	╩	234	Û
11	♂	43	+	75	K	107	k	139	ï	171	½	203	╦	235	Ù
12	♀	44	,	76	L	108	l	140	î	172	¼	204	╠	236	ý
13	♪	45	-	77	M	109	m	141	ì	173	¡	205	═	237	Ý
14	♫	46	.	78	N	110	n	142	Ä	174	«	206	╬	238	¯
15	☼	47	/	79	O	111	o	143	Å	175	»	207	¤	239	´
16	►	48	0	80	P	112	p	144	É	176	░	208	ð	240	-
17	◄	49	1	81	Q	113	q	145	æ	177	▒	209	Đ	241	±
18	↕	50	2	82	R	114	r	146	Æ	178	▓	210	Ê	242	=
19	‼	51	3	83	S	115	s	147	ô	179	│	211	Ë	243	¾
20	¶	52	4	84	T	116	t	148	ö	180	┤	212	È	244	¶
21	§	53	5	85	U	117	u	149	ò	181	Á	213	ı	245	§
22	▬	54	6	86	V	118	v	150	û	182	Â	214	Í	246	÷
23	↨	55	7	87	W	119	w	151	ù	183	À	215	Î	247	¸
24	↑	56	8	88	X	120	x	152	ÿ	184	©	216	Ï	248	°
25	↓	57	9	89	Y	121	y	153	Ö	185	╣	217	┘	249	¨
26	→	58	:	90	Z	122	z	154	Ü	186	║	218	┌	250	·
27	←	59	;	91	[	123	{	155	ø	187	╗	219	█	251	¹
28	∟	60	<	92	\	124	¦	156	£	188	╝	220	▄	252	³
29	↔	61	=	93	]	125	}	157	Ø	189	¢	221	¦	253	²
30	▲	62	>	94	^	126	~	158	×	190	¥	222	Ì	254	■
31	▼	63	?	95	_	127	⌂	159	ƒ	191	┐	223	▀	255	